

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

SIMPLE: A PROTOTYPE SOFTWARE FAULT-INJECTION TOOL

by

Neil John P. Acantilado
Christopher P. Acantilado

December 2002

Thesis Advisor:
Second Reader:

J. Bret Michael
Richard H. Riehle

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 2002	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: SIMPLE: A Prototype Software Fault-Injection Tool			5. FUNDING NUMBERS	
6. AUTHOR(S) Neil John P. Acantilado & Christopher P. Acantilado				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) Fault-injection techniques can be used to methodically assess the degree of fault tolerance afforded by a system. In this thesis, we introduce a Java-based, semi-automatic fault-injection test harness, called Software Fault Injection Mechanized Prototype Lightweight Engine (SIMPLE). SIMPLE employs a state-based fault injection approach designed to validate test suites. It also can assist developers to assess the properties of a system such as robustness, reliability, and performance. SIMPLE employs fault acceleration to test a system's fault-tolerant capabilities. We present an object-oriented analysis of the system and several case studies, using software fault injection on specific, targeted systems, to assess SIMPLE's effectiveness.				
14. SUBJECT TERMS Software Fault Injection, Fault Tolerance, Software Testing, Software Test Coverage, and Metrics			15. NUMBER OF PAGES 226	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SIMPLE: A PROTOTYPE SOFTWARE FAULT-INJECTION TOOL

Neil John P. Acantilado
B.A., University of California San Diego, 1992

Christopher P. Acantilado
B.S., San Diego State University 1993

Submitted in partial fulfillment of the
Requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
December 2002**

Authors: Neil John P. Acantilado

Christopher P. Acantilado

Approved by: J. Bret Michael
Thesis Advisor

Richard Riehle
Second Reader

Valdis Berzins, Chairman
Software Engineering Curriculum

Chris Eagle, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Fault-injection techniques can be used to methodically assess the degree of fault tolerance afforded by a system. In this thesis, we introduce a Java-based, semi-automatic fault-injection test harness, called Software Fault Injection Mechanized Prototype Lightweight Engine (SIMPLE). SIMPLE employs a state-based fault injection approach designed to validate test suites. It also can assist developers to assess properties of a system such as robustness, reliability, and performance. Furthermore, SIMPLE employs fault acceleration to test a system's fault-tolerant capabilities. We present an object-oriented analysis of the system and several case studies, using software fault injection on specific, targeted systems, to assess SIMPLE's effectiveness.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	PROBLEM STATEMENT	1
B.	RESEARCH ISSUES	2
1.	Identifying SWFI Impact	2
2.	Java Programming Language	2
3.	Identifying Metrics	2
4.	Selecting a Methodology	3
5.	Faults Models	3
6.	Evaluating SIMPLE	3
C.	RESULTS AND CONCLUSIONS.....	3
II.	SOFTWARE FAULT INJECTION.....	5
A.	PURPOSE.....	5
B.	BENEFITS.....	5
1.	Fault Acceleration.....	5
2.	COTS Testing	5
3.	Increases Test Coverage	6
4.	Sensitivity Analysis	6
C.	LIMITATIONS	6
D.	SWFI TESTING.....	7
1.	Code Mutation	7
2.	Data Mutation.....	7
E.	SWFI TECHNIQUES	8
1.	Software Trap	8
2.	Meta-object Protocol	8
3.	Wrapper	9
4.	Perturbation Functions.....	9
5.	Interface Mutation.....	9
6.	Assertion Violation	10
7.	Messaging Oriented Middleware (MOM)	10
III.	METRICS	13
A.	SOFTWARE METRICS.....	13
1.	SWFI Metrics.....	13
a.	<i>Fault Coverage.....</i>	<i>13</i>
b.	<i>Code Coverage.....</i>	<i>13</i>
c.	<i>Test Adequacy</i>	<i>14</i>
d.	<i>Sensitivity Analysis via the PIE model</i>	<i>14</i>
IV.	ISSUES AND CHALLENGES	15
A.	INSTRUMENTATION/OVERHEAD.....	15
B.	COMPILE TIME VS. RUN TIME.....	15
C.	LATENT FAULTS.....	16
D.	FAULT SIMULATION	16
E.	FAULT PROPAGATION.....	16

V.	RELATED RESEARCH	19
A.	SWFI TOOLS.....	19
B.	FERRARI	19
C.	XCEPTION	19
D.	GOOFI.....	20
E.	DOCTOR.....	20
F.	FAILURE SIMULATION TOOL (FST)	21
G.	FTAPE	22
H.	IDEAS FOR SWFI DEVELOPMENT	22
VI.	SIMPLE – DESIGN AND IMPLEMENTATION	23
A.	INTRODUCTION	23
B.	ACTIVITY AND CLASS DIAGRAMS	24
C.	FAULTS IN SIMPLE.....	25
	1. Fault Types.....	26
	2. Fault Attributes	27
	3. Fault Triggers	29
D.	JAVA TECHNOLOGIES	29
	1. Candidate Technologies.....	30
	2. Selected Technologies	34
E.	STATE AND SEQUENCE DIAGRAMS	37
F.	ISSUES, CAVEATS, LIMITATIONS, AND LESSONS-LEARNED	39
	1. Java Virtual Machine (JVM) Compatibility.....	39
	2. Requires Compilation with Debug Option	39
	3. How to Handle Fault-Injection Time?	39
	4. Pre-instrumentation – A Necessary Evil.....	42
	5. Pre-instrumentation Behavior is Inconsistent	42
	6. Requires Source Code and Strong Familiarity Thereof.....	43
	7. Heisenberg’s Uncertainty Principle	44
	8. Affect of Compiler-Induced Optimizations	46
	9. Software Fault Evaluation is Coarse-Grained.....	46
	10. Interaction with Other Software Tools	47
G.	FUTURE ENHANCEMENTS.....	48
	1. Using Perturbation Functions to Improve Variable-Mutation Performance.....	48
	2. Extending the Fault Range of SIMPLE	49
	3. Mutating Collections and Arrays.....	50
	4. Data Collection in SIMPLE.....	50
	5. Developing a GUI for SIMPLE	50
	6. Further Investigation of Java Technologies.....	51
	7. Further Investigation of Open-Source Fault-Injection Tools	51
H.	PLAN TO THROW ONE AWAY	52
VII.	CASE STUDIES	53
A.	INTRODUCTION	53
B.	CASE STUDY I: USING SIMPLE TO VERIFY TEST CASES	53
	1. CSMA/CD Software Description	54
	2. JUnit Framework	54

3.	CSMA/CD Test Suites	55
4.	CSMA/CD Test Cases.....	57
5.	Employing Fault-Injection	61
6.	Results	66
7.	Discussion.....	66
C.	CASE STUDY II: UNCOVERING SOFTWARE ANOMALIES USING SIMPLE.....	67
1.	The Airline Reservation System (ARS) Software Description	68
2.	Testing the ARS Exception-Handling Capabilities	69
3.	Assessing GUI Performance via Fault-Acceleration	76
4.	Discussion.....	79
D.	CASE STUDY III: INCREASING TEST COVERAGE	80
1.	Coverage Metrics	80
2.	Gretel.....	81
3.	Using Gretel with SIMPLE	81
4.	Assessing SIMPLE Coverage	83
5.	Results	83
6.	Discussion.....	87
VIII.	CONCLUSION	89
IX.	LIST OF REFERENCES	91
	APPENDIX A – SIMPLE UML DIAGRAMS	95
	APPENDIX B – FAULT SPECIFICATION GRAMMAR.....	105
	APPENDIX C – CASE STUDY UML DIAGRAMS.....	109
	APPENDIX D – CASE STUDY FAULT CONFIGURATION FILES	113
	D-1 CSMA/CD UNIT-TEST FAULT CONFIGURATION FILE (CASE STUDY I).....	114
	D-2 ARS FAULT CONFIGURATION FILE (CASE STUDY II, PART 1)	117
	D-3 ARS FAULT CONFIGURATION FILE (CASE STUDY II, PART 2)	118
	D-4 GRETTEL/ARS FAULT CONFIGURATION FILE (CASE STUDY III)..	119
	APPENDIX E – SIMPLE SOURCE CODE.....	121
	E-1 BUILD.XML	122
	E-2 DOM_UTIL.JAVA	124
	E-3 EVENTTHREAD.JAVA.....	127
	E-4 FAULT.JAVA	136
	E-5 FAULTMANAGER.JAVA.....	143
	E-6 FAULTPARSER.JAVA.....	149
	E-7 LOCATIONFAULTTRIGGER.JAVA	157
	E-8 OBJECTFAULT.JAVA.....	159
	E-9 OBJECTFIELDFAULT.JAVA	162
	E-10 OBJECTLOCALFAULT.JAVA	166
	E-11 PRIMITIVEFAULT.JAVA	168
	E-12 PRIMITIVEFIELDFAULT.JAVA	170
	E-13 PRIMITIVELOCALFAULT.JAVA	174

E-14 SIMPLEHARNESS.JAVA	176
E-15 SIMPLEHELPER.JAVA.....	181
E-16 SIMPLEREPOSITORY.JAVA	187
E-17 SIMPLETREK.JAVA.....	189
E-18 STARTTIME.JAVA	194
E-19 STATEMENTHELPER.JAVA	195
E-20 STREAMREDIRECTTHREAD.JAVA	201
E-21 UPDATETIME.JAVA	203
E-22 UTIL.JAVA.....	204
E-23 UTILITYASPECT.JAVA.....	207
INITIAL DISTRIBUTION LIST	209

LIST OF FIGURES

Figure 1. Sample Fault Input	26
Figure 2. VerboseMetaobject class	31
Figure 3. Example Advice	32
Figure 4. Arbitrary Code Segment	36
Figure 5. Application that is Activated by the RUN Button.....	41
Figure 6. Code Snippet	45
Figure 7. Perturbation Function	48
Figure 8. JUnit GUI.....	55
Figure 9. SimpleTest – Tests All Passed	61
Figure 10. SimpleTest – Tests All Failed	66
Figure 11. ARS Source Code Snippet	70
Figure 12. Flight Manager GUI	71
Figure 13. Travel Agent Reservation GUI with Reservation Dialog Box	72
Figure 14. The Flight Manager GUI Session with Injected Fault.....	73
Figure 15. A Travel Agent Reservation GUI Session with Injected Fault	74
Figure 16. Reservations in the Travel Agent Reservation GUI	75
Figure 17. Software Bug in the Travel Agent Reservation GUI Code.....	75
Figure 18. Template Fix	76
Figure 19. The Travel Agent Reservation GUI.....	77
Figure 20. Partial Listing of the Insert Method.....	77
Figure 21. A "Frozen" Travel Agent Reservation GUI.....	78
Figure 22. Instrumenting with Gretel.....	82
Figure 23. Querying All Flight Data in ARS.....	82
Figure 24. Session 1 Coverage Results	84
Figure 25. Session 2 Coverage Results	86
Figure A-1. SIMPLE Activity Diagram	95
Figure A- 2. High-level Class Diagram of SIMPLE Components	96
Figure A-3. Detailed Class Diagram of SIMPLE Components.....	97
Figure A-4. General State Diagram of SIMPLE Processes	98
Figure A-5. High-level Sequence Diagram of SIMPLE Process.....	99
Figure A-6. Sequence Diagrams of Faults Being Parsed	100
Figure A-7. Sequence Diagram of Classes Being Prepared	101
Figure A-8. Sequence Diagram of Fault Triggers Being Served.....	102
Figure A- 9. State Diagram of SUT Instrumentation	103
Figure C- 1. Class Diagram for the CSMA/CD Simulation Software	109
Figure C- 2. JUnit Framework Class Diagram	110
Figure C- 3. Test Suite Class Diagrams	111
Figure C- 4. Class Diagram for the ARS System	112

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Test Suite Descriptions	56
Table 2. "NetworkSimulationMainTest" Test Cases	57
Table 3. "PacketQueueTest" Test Cases.....	58
Table 4. "StationTest" Test Cases	59
Table 5. "NetworkEventManagerTest" Test Cases.....	60
Table 6. "NetworkTest" Test Cases	60
Table 7. "NetworkSimulationMainTest" Faults.....	62
Table 8. "PacketQueueTest" Faults.....	63
Table 9. "StationTest" Faults	64
Table 10. "NetworkEventManagerTest" Faults	64
Table 11. "NetworkTest" Faults.....	65

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Our sincerest appreciation to

Dr. J. Bret Michael & Mr. Richard Riehle

for their guidance and support in this endeavor.

Our deepest gratitude to

Nancy W. Chang, M.D.

for her diligence in proof-reading our thesis and other similar work. During these past few years, we listened to and incorporated many of her comments and suggestions.

We became better writers for it.

And, thanks to

Mom & Dad

for their love and support.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

System applications play a major role in today's society. Many commercial industries and government agencies are developing critical systems to satisfy people's everyday needs. Such systems include power management, medical devices monitoring, and transportation scheduling. For example, a simple automatic coffeemaker can be categorized as a critical system. Unfortunately, unexpected system failures can have serious consequences, such as loss of life or property, damage to the environment, or denial of service.

Emphasis has traditionally been placed on planning and executing testing activities late in the software-development process. However, unforeseen catastrophic disasters caused by latent software errors cannot with certainty be anticipated [11]. One way to safeguard against the effects of software defects is to design fault tolerance into systems.

Automatic fault-injection tools, techniques and methodologies exist for assessing the robustness and reliability of systems [20]. For example, researchers at Cigital (formerly known as Reliable Software Technologies) have developed a tool that programmatically invokes system-level exceptions used to evaluate the effects of Windows NT failures in an application [1]. Results gathered from these tests assist developers in strengthening exception handling, identifying essential pre-condition assertions, and widening software test coverage of the system.

We built a semi-automatic fault-injection test harness geared towards testing system applications. In order to realize automated testing, we implemented our fault-injection engine to be non-destructive to the application source, with the realization that some byte-code instrumentation of the Software-Under-Test (SUT) may be necessary.

The purpose for this research is to investigate how a fault injection test harness can be designed to accommodate the software test process for safety-critical applications.

B. RESEARCH ISSUES

Key research issues are summarized below.

1. Identifying SWFI Impact

The intrusiveness of software fault injection (SWFI) techniques can adversely affect software behavior. For instance, both performance and resource overhead can be incurred during SWFI testing. We discuss the impact that SWFI can have on the SUT.

2. Java Programming Language

Like many commercial industries, the DOD is turning to Java to develop many of its mission-critical systems (e.g., DIICOE¹). Also, many of these systems consist of COTS components, written in Java, used to support Joint operations.² For instance, approximately 50% of the DIICOE kernel is comprised of Java technology alone [2]. Major projects at our facility³ use DIICOE as the underlying kernel for many of its command and control systems under development (e.g., GCCS-M⁴). For these reasons, this study will focus on developing a SWFI tool that is geared towards Java-based systems. In addition, existing Java technologies will facilitate our fault-injection tool development.

3. Identifying Metrics

Many existing SWFI tools compute metrics [1, 16, 17, 18, 21, 22, 25, 26, 28, 32]. For example, Fault Tolerance and Performance Evaluator (FTAPE) computes the error detection rate as the ratio of detected errors to injected faults; error recovery is measured as the number of system crashes; and error recovery is measured by performance degradation [32].

¹ Defense Infrastructure Information/Common Operating Environment (DIICOE) is an architecture that provides a common runtime environment for Command & Control, Communications, Computers, and Intelligence (C4I) systems

² Joint Operations is the unification of actions between the Armed Forces of the United States. For more information, see website available at http://www.dtic.mil/doctrine/jfe_briefing_modules.htm, October 2002.

³ By facility, we mean our place of work at the Space And Naval Warfare Systems Center in San Diego (SSCSD). For more information see website available at <http://www.spawar.navy.mil/>, August 2002.

⁴ Global Command And Control System Maritime (GCCS-M) provides C4I services to the fleet giving allied maritime forces the ability to operate in a network-centric environment. See website available at <http://jite.fhu.disa.mil/gccsiop/interfaces/gccsm.htm>, May 2002.

Since SWFI plays a significant role in software test coverage (a metric for test adequacy of test schemes), our case study discusses how SIMPLE can increase test coverage.

4. Selecting a Methodology

We selected candidate fault models, attributes and methodologies, and applied them in concert with our SWFI tool as a means for testing the effectiveness of SIMPLE.

5. Faults Models

Many SWFI tools employ built-in fault models to be used to assess the robustness of the SUT. Likewise, SIMPLE has its own set of faults to inject into a targeted system: exceptions, data mutation, time delays, and memory leaks.

6. Evaluating SIMPLE

We experimented with SIMPLE via a case-study approach. Each case study includes a discussion on relevant tool features, associated fault models, pertinent test results, and lessons learned (covering the implications and limitations of the test harness). We also discuss the conclusions reached and experiences gained as a result of this study.

C. RESULTS AND CONCLUSIONS

After thorough research, we were able to construct a SWFI test-harness for Java-based systems. Ideas were borrowed from existing SWFI tools and then incorporated into our SWFI prototype. Our development was facilitated through use of available open-source Java technologies, such as Sun's Java Platform Debugger Architecture (JPDA) and Compaq's JTrak APIs.

Even in its early form, our prototype proved to be very promising as depicted in our case studies. See Chapter VII.

THIS PAGE INTENTIONALLY LEFT BLANK

II. SOFTWARE FAULT INJECTION

A. PURPOSE

Rather than exhaustively search for faults, one can directly inject simulated faults into a system and then analyze the effects of injected faults. SWFI dynamically demonstrates whether the system tolerates improbable inputs or outputs, such as the database overflow error suffered by the Aegis Missile Cruiser, USS Yorktown: the database error caused the jet propulsion system to shutdown, thus, leaving ship crippled for hours [4].

B. BENEFITS

Unexpected system failures stemming from inadequate testing practices can have serious implications and consequences for safety-critical systems (e.g., Therac 25 [5], Ariane 5 Flight 501 [66]) [3]. Thus, organizations should be fully aware of the advantages and limitations of SWFI in detecting software errors.

1. Fault Acceleration

One advantage SWFI provides is that it encourages fault acceleration. Rather than investing time testing for failure occurrences in a system, faults are intentionally injected in a desirable time frame [7]. Then their effects are analyzed. This process is known as fault acceleration.

2. COTS Testing

Critical systems increasingly utilize Commercial-Off-The-Shelf (COTS) software during development. Flaws in these systems can create intolerable losses. Unfortunately, COTS makes it increasingly difficult to test underlying features, such as error/exception handling routines, due to source code unavailability.

Fortunately, organizations are now required to openly provide APIs to their customers [8]. In addition, more open-source software applications exist. As a result, researchers are using fault injection as a means to effectively test COTS software. For example, developers inject faults into the underlying operating system or processing hardware of the system [9].

Furthermore, if application source is unavailable, we can conduct COTS testing with the traditional, black-box testing approach using SWFI. In other words, SWFI will inject faults into the software's known entry points (i.e., interfaces, public methods).

3. Increases Test Coverage

In a recent study, researchers have determined that fault injection increases test coverage of the software [10]. Fault injection forcefully executes difficult to reach paths in the program. For example, fault injection techniques can violate explicit assertion statements. This allows for the forceful execution of hard to reach areas of the program. According to a recent case study, SWFI increased test coverage by as much as 90 percent [10].

4. Sensitivity Analysis

Sensitivity Analysis predicts where faults will hide, especially from test cases. In essence, Sensitivity Analysis helps to measure software testability. Sensitivity analysis involves three separate processes: execution, infection, and propagation analysis. To perform the latter two analyses, fault injection techniques must be used. Infection and propagation analysis require mutation of software and its subsequent internal states created during run-time. Hence, researchers are employing SWFI effectively perform Sensitivity Analysis [30, 31]. Sensitivity Analysis is briefly discussed in Chapter III.

C. LIMITATIONS

Although SWFI provides benefits towards software testing, there exist limitations. For example, some SWFI tools require code instrumentation [28]. This may cause unnecessary overhead to the system's performance. As a result, code that is running during testing will not be necessarily the same code running at a realistic environment. Other limitations include the inability for SWFI to mimic fault latency and fault propagation. Therefore, software developers and testers should not rely solely on SWFI for accurate testing. See Chapter IV for more discussion on SWFI limitations.

Furthermore, SWFI focuses on determining how software behaves in the presence of a range of faults produced in a non-ideal environment [3]. It is incapable of fully assessing software correctness as it relates to compliancy to requirements. Hence, SWFI

is best served as a compliment, rather than a replacement, to traditional software testing techniques.

As Dr. Bret Michael, Professor of the Naval Postgraduate School, states, “Testing is never complete because there are more possible states that a system can enter that you can possibly test for. You need multiple approaches to assessing the pedigree of the software. You don’t want to put all your eggs in any one of those baskets of techniques. You need to have *cross-cutting* techniques that are feasible approaches [29].”

D. SWFI TESTING

Mutation testing, a common SWFI test, detects the differences between the application’s intended behaviors and its newly changed behaviors. It analyzes how the resultant changes affect the application’s software testability.⁵ Typical mutation tests include direct application of fault injections to existing source code (referred as “code mutation”) or dynamic insertion during system execution (referred as “data mutation”). The following subsections describe the basics of code and data mutation.

1. Code Mutation

Code mutation is the process of directly changing existing source code. Its purpose is to change the state of the executing program. The modified code is termed a mutant. As a simple example, consider the following code statement, $a = a + 1$. Through SWFI, this statement can be changed to $a = a + a + 1$ or $a = a + 10$. There are different levels to code mutation [3]. The example just described is classified as a first-order mutant. A second order mutant is achieved by mutating a first order mutant. A third order mutant requires the mutation of a second order mutant, and so forth.

2. Data Mutation

Data mutation modifies the program’s internal state (e.g., memory, time, variables) at runtime. Overriding programmer-defined variables or the data transferred via function calls causes this modification. Data mutation is the preferred method of mutation testing, mainly because we are largely concerned with internal data states that cause failures [11].

⁵ Testability is a software characteristic that measures its ability to detect faults during test time.

E. SWFI TECHNIQUES

Many researchers and engineers have developed novel SWFI mechanisms. This section briefly describes some of their approaches.

1. Software Trap

Software traps are instructions that can be placed anywhere in the target program. When detected by a processor, the software trap halts execution of the current process. Software traps are triggered either by program execution or by a timer. They are particularly useful for injecting CPU, memory, and bus type of faults [20].

Software traps can be used to trigger faults via fault injection. For example, FERRARI, a SWFI tool, uses software traps to invoke faults physically into the system via SWFI [28]. Specifically, FERRARI sets a trace bit (i.e., software trap) into the target program process control block. When the target program process reaches the trace bit, a context switch is made to the fault injection process. The fault injection process then alters the program state by executing a sequence of system calls.

2. Meta-object Protocol

The Java programming language supports a *Reflection* API that enables the ability of a program to introspect its own behavior.⁶ In other words, the program will be able to discover information about any Java class, including its set of methods and corresponding parameterized types.

Java's Reflection API has been extended to allow for a program to alter its own behavior. This extension is commonly known as behavioral reflection [12]. Using both a meta-object⁷ protocol [13, 14] and behavioral reflection can dynamically capture an operation (or method invocation), alter it, and execute it. Technically speaking, the run-time system invokes a meta-object method that is associated with a particular operation. The logic of the meta-object method is pre-instrumented by the developer to reflect a changed behavior of an operation. For example, a meta-object method, *subtract*, is executed whenever an *add* function is called. This capability makes reflection a well-suited mechanism for SWFI [14].

⁶ Reflection enables the programmatic identification of class and object information during run-time.

⁷ Metaobjects encapsulates the behavior adaptations of a component.

3. Wrapper

A wrapper encases a component and the operations that it provides. Specifically, the wrapper captures calls made to the encased component or shields the system from certain component outputs. Wrappers are heavily used to test COTS components [15].

Wrappers can serve as a mechanism for fault injection. In particular, these components can produce exceptions and error conditions when particular system functions are invoked during execution. For example, a wrapper component can intercept a system call and then change intended behavior by mutating its inputs. The Fault Simulation Tool (FST) uses this approach to evaluate the robustness of the Windows NT platform. The tool utilizes Win32 dynamic linked libraries (DLL) that are wrapped to support the corruption of DLL input data on demand. Refer to [1] for more details concerning the FST tool.

4. Perturbation Functions

Perturbation (or perturb) functions are used to forcefully override the current internal value of a variable, thereby simulating errors. For example, a random function generator can be used as a perturb function. Thus, the statement, $a = a + 1$, can be changed to $a = rand(a) + 1$.

Unfortunately, perturbation functions are usually applied at a source code level (i.e., the function is compiled into the targeted system). To avoid being intrusive to the application source, these functions can be applied at the byte-code level. For example, a perturb function, *flipBit*⁸, which can exist in a separate program, but must be linked with the targeted program's executable.

5. Interface Mutation

System of systems is a composition of individual systems, which are organized to achieve a common goal (e.g. airports which consist of aircraft, terminals, runways, air traffic controls and baggage handling systems). Unfortunately, such systems are prone to errors that propagate across system boundaries. For example, data that is corrupt in the message traffic may cause a system to perform an inadvertent system operation. Hence,

⁸ A flipbit function simply flips a bit from 0 to 1 and vice versa. Flipping bits encourages simulation of many flaws (e.g., data corruptions).

proper interface testing is required to see whether systems could handle receiving corrupted data.

A technique designed to test at the system of systems level is Interface Mutation [19]. Like mutation testing, this method is designed to create mutants by making changes to entities. However, unlike mutation testing, only those entities that reside on interfaces between components are mutated. This helps to limit the number of mutants to execute and analyze. The entities between interfaces include: function calls, function return values, and global data shared by two or more functions. Thus, mutating entities between interfaces can stimulate errors. For example, a function directly relating to two components can be called with either a missing or incorrect parameter.

For a list of other potential errors injected at the interface level, see [33].

6. Assertion Violation

Assertions are boolean expression constructs that specify a program's expected behavior. Examples of such constructs include pre-conditions, post-conditions, and class-invariants. Specifically, an assertion about the program's current state must be true before, during, and after a function is invoked. Thus, certain boolean conditions must be satisfied before an operation can be carried out. A true assertion statement ensures that a function is executed correctly; whereas, a false assertion statement guarantees a fault.

Researchers in [10] are using assertions as a technique for injecting software faults. To simulate a fault, an assertion is made false during program execution in an automatic fashion. This allows for the modeling and simulation of faults. Examples of such faults include assignment, function, and initialization faults. Furthermore, invalid assertions can cause a chain of other assertion violations in the code. In effect, this increases test coverage by exercising the assert mechanisms.

7. Messaging Oriented Middleware (MOM)

As discussed in the previous section, organizations are integrating their applications into a single enterprise-wide system. Many of these systems use MOM to handle the exchange of information (or messages) across a distributed system. For

example, existing MOM systems utilizing JMS⁹ or SOAP¹⁰, offer communication services such as point-to-point and publish-and-subscribe messaging.

Unfortunately, problems can arise when the underlying MOM mechanism starts to scale up as a result of increasing client connections. Typical problems include: network bottlenecks, memory consumption, threading contention, data loss, message congestion, low disk space, and untimely message delivery. Such adverse effects are not acceptable, especially, for systems having safety- or mission- critical like properties. Therefore, stress testing against the system is necessary to ensure robustness.

Fortunately, useful SWFI techniques can be employed using basic MOM constructs. For example, multiple senders (or producers), multiple receivers (or consumers), or both can be easily instantiated to stress test system load. The system load potentially causes it to behave differently.

Another form of SWFI-based stress test is the emulation of message congestion. Developers can implement producers designed to "inject" multiple messages at high frequency causing adverse effects against the system such as in-memory buffer (or queue) overflow.

Finally, message corruption is another practical SWFI technique that MOM can simulate. By using brute-forcing producers to send invalid messages, developers or testers can assess the system's fault tolerance capabilities (i.e., exception-handling).

⁹ Java Messaging Service (JMS) is a Java Standard API designed to implement MOM systems.

¹⁰ Simple Object Access Protocol (SOAP) is an XML-based web services MOM protocol that facilitates web servicing.

THIS PAGE INTENTIONALLY LEFT BLANK

III. METRICS

A. SOFTWARE METRICS

Software metrics are statistical data used to evaluate the properties of the software. For example, the Lines-Of-Code (LOC) is a commonly used software metric that measures the size of the software. Another example is the Function Point (FP) Size Estimation metric. It measures the complexity of the functions used in the software.

Many of these metrics are correlated to the development effort of the software. However, they are not necessarily correlated to the properties of software systems that are of interest from a test and evaluation perspective.

1. SWFI Metrics

SWFI has been used as a means to measure the desired properties of the software such as vulnerability, robustness, and survivability [9, 23, 24, 25]. One analyzes and collects error-based metrics during SWFI testing, such as the number of abnormal exits or the number of system crashes. The following subsections cover some of the metrics that could be used to quantitatively evaluate results obtained from SWFI testing; other specialized SWFI metrics are listed in [3].

a. Fault Coverage

A failure-based (or fault-based) strategy as it pertains to fault-injection attempts to measure the fault-tolerant characteristics of an application. This measurement is specifically referred to as the *fault-coverage* metric of the application. Ghosh, Mathur, and Horgan define fault coverage as “... the percentage of the number of faults tolerated with respect to that of faults injected [33].” In this scheme, contextually relevant faults are programmatically injected into the application. In accordance, the tester then evaluates and records any application responses to the fault. A case study that utilizes a failure-based fault injection approach is given in [33].

b. Code Coverage

Code coverage is often used as an exit criterion for software testing. In many cases, testing is deemed to be complete once a threshold coverage value has been

met. The number of statements or branches exercised during testing typically delineates this threshold value. Fault-injection can be used to increase coverage by executing those “hard-to-reach” software paths [10]. Interestingly enough, the application’s exception-handling and error-recovery mechanisms are typically the most inaccessible areas to reach during testing.

c. Test Adequacy

To measure the effectiveness of application test cases, the combination of fault coverage and code coverage metrics forms a two-dimensional metric [33]. This metric provides an effective *test adequacy* measurement. For instance, a low score for both fault coverage and code coverage yields a poor test-adequacy rating, as does a high fault-coverage score with a low code-coverage score.

d. Sensitivity Analysis via the PIE model

Sensitivity analysis utilizes fault injection to predict where in the source code test cases will be incapable of revealing errors [3]. Hence, it also purports to define a kind of test-adequacy metric. In his PIE model, Voas proposes a sensitivity analysis approach for deriving various prediction measurements that relate fault-sensitivity to the software. More specifically, PIE is comprised of three separate analyses known as the propagation (i.e., determines the likelihood that a data state error propagates to the output space), infection (i.e., measures the likelihood of corrupted internal states), and execution analysis (i.e., estimates the likelihood of code execution at each location). Each of these analyses contributes to a metric that determines the likelihood that faults will be uncovered within an application during software testing. See [3] for complete details.

IV. ISSUES AND CHALLENGES

A. INSTRUMENTATION/OVERHEAD

Common SWFI mechanisms such as perturbation functions require the modification of the program. Unfortunately, this extra instrumentation causes execution overhead that will affect system behavior such as performance [16]. Furthermore, existing SWFI tools require their processes to be executed separately or during the target system's process. For example, the SWFI tool, FERRARI, requires some context switches¹¹ between its fault injection process and the target system process. However, this requirement creates a timing overhead that can also adversely change the behavior of the SUT. This behavioral effect is commonly known as a *Heisenbug*¹², software's rendition of the *Heisenberg Uncertainty Principle*.¹³ In other words, intrusiveness of software instrumentation can alter the behavior of the software under test.

B. COMPILE TIME VS. RUN TIME

Software fault injection can be categorized according to when an injection is performed [20]. For example, a fault injection can occur during compilation or runtime. Compile-time injection modifies code instructions in the program execution. In contrast, run-time injection requires a mechanism (i.e., an injector) to inject faults when the program is running. In addition, the program must be prepared before performing a fault injection experiment. However, each approach has its advantages and disadvantages, as discussed below.

Low intrusiveness can be achieved via compile-time injection. In this method, no control is required to run the fault-injection experiment at run-time. Moreover, no perturbation is introduced in the SUT during its execution. However, since there is no control, there is no way to tell whether a fault was activated or has affected the software under test.

¹¹ Context switch is the process of switching between one process to the kernel and vice versa.

¹² The term Heisenbug was originally derived from the Heisenberg Uncertainty Principle. Heisenbugs are intermittent software faults that are not necessarily guaranteed to produce an error based on deterministic inputs. More notably, they are often very hard to locate [45].

¹³ The Heisenberg Uncertainty Principle is the inability to simultaneously measure conjugate attributes, such as position and momentum, of a subatomic particle.

On the contrary, run-time injection can present a high level of intrusiveness; an extra mechanism is needed to inject faults into the SUT. Furthermore, there must be a way for the program to invoke or “trigger” the fault to be injected. The downfall of this method is that it requires instrumentation that will eventually affect the system’s behavior.

C. LATENT FAULTS

The system's hardware or operational behavior is vulnerable to actual hardware or software faults that affect memory, clock value, control flow, and so on. Moreover, they may lie dormant and undetected for hundreds of thousands of hours of operation. For example, failures in system memory may not be apparent until faults have occurred in the CPU's circuitry long before. Although, SWFI is ideal in representing memory faults, SWFI cannot mimic latent faults. Thus, SWFI may fail to capture certain behaviors caused by latent faults. Proper hardware monitoring can solve this issue. However, this solution incurs overhead on the SUT.

D. FAULT SIMULATION

Exhaustive testing is impossible to achieve except in trivial cases [27]. Exhaustive testing requires a test suite to test for all possible inputs and states. Similarly, the approach to inject every fault that targeted system may face is infeasible. This assumption is due to the fact that the anomaly space of the targeted system can be infinitely large [6]. Thus, system analysts would have to go through a time-consuming process of determining faults that are likely to be encountered by the targeted system during its lifetime. Unfortunately, analysts may accidentally foresee many faults that lead to failures of much more importance.

E. FAULT PROPAGATION

The test results obtained by SWFI can give a rudimentary assessment of the robustness of a system. However, results captured from SWFI experiments are observed and captured at the final impact of the system. Thus, it is not clear what actually happens after a fault was injected or whether or not a fault propagates in the SUT after injection. Moreover, most of the existing SWFI tools lack the ability to produce those faults that propagate and result in unexpected future behavior (i.e., race condition) in the system.

The majority of faults that SWFI tools are capable of simulating are basic in nature. The faults are typically due to coding errors, I/O errors, and memory corruption. Fortunately, researchers are now beginning to research how SWFI can be used to assess fault propagation [26].

THIS PAGE INTENTIONALLY LEFT BLANK

V. RELATED RESEARCH

A. SWFI TOOLS

Today's critical systems designed to satisfy people's demands, require thorough testing of many of their essential system attributes, such as reliability, availability, and safety. This has given rise to different approaches of implementing fault injection tools. The following briefly describes some of the existing SWFI tools, including some of the features that were considered for our study.

B. FERRARI

The Fault and Error Automatic Real-time Injector (FERRARI) tool emulates hardware faults using software traps [28]. These traps inject CPU, memory, and bus fault types. Two running concurrent processes carry out the actual fault-injection process: fault/error injection process and the target program process.

The fault/error injection process begins by having the target program process set itself as traceable. Later, the target program loads itself into memory and starts its execution. The target program then encounters a software trap; this occurs when a trace bit is encountered from the process control block. The software trap then invokes the fault/error injection process to execute a sequence of system calls used for mutation purposes (e.g., altering the content of memory and registers). Typically, fault injections involve altering content from selected registers or memory locations.

C. XCEPTION

XCEPTION uses a debugging and monitoring model approach to inject faults into software [16]. The debugger is directly programmed into the hardware to allow for the complete separation between the SUT and the fault injector. As a result, code instrumentation is avoided and the tool's fault injection process can take advantage of already defined fault triggers (i.e., hardware exception triggers) in the processing hardware.

Unlike FERRARI, XCEPTION does not use software trap instructions to trigger fault injection. Rather, it uses a processor's built-in hardware fault triggers to invoke fault injection. The fault injector is implemented as an exception handler. Thus, when

XCEPTION reaches a predetermined accessible address, an exception is raised, and a fault is injected. As a result, the SUT's memory content and registers are corrupted according to the specific fault type.

D. GOOFI

GOOFI (Generic Object-Oriented Fault Injection Tool) is a platform independent tool that provides a user-friendly SWFI environment [21]. GOOFI's main purpose is to provide support for the adaptation of new fault injection techniques. Many of its building blocks consist of abstract methods that are reusable when defining algorithms from other SWFI techniques.

Currently, GOOFI supports a SWFI technique called Scan-Chain Implemented Fault Injection (SCIFI). The SCIFI technique uses built-in logic¹⁴ to inject faults into pins and other internal elements of an integrated circuit. The SCIFI fault injection process begins when the SUT is fully initialized with workload information and initial inputs, such as *campaign data*.¹⁵ In addition, the fault injection algorithm reads campaign data from an SQL database. The user is responsible for providing this information via a GUI. The database stores all SWFI data (e.g., targeted system information, fault injection experiments).

When a breakpoint condition is reached, chosen faults are injected by reading scan-chains, bits are inverted, and the resulting scan-chains are written back to the system. The whole process repeats itself until a termination condition is reached. Throughout the entire fault injection process, system states were captured and stored into the database for further analysis. Results typically include: detected and escaped errors, latent errors, and overwritten errors (i.e., no difference of system dates between pre-fault injected state and post-fault injected state).

E. DOCTOR

DOCTOR was designed to address the inability of SWFI to emulate the effects of actual faults (e.g., latent faults commonly caused by communication errors) [22].

¹⁴ An example of a built-in test-logic is the boundary scan-chains and internal scan-chains that are present in many modern VLSI circuits.

¹⁵ As it pertains to SWFI, campaign data can consist of fault models, fault injection breakpoints, injection times, and bit inversions.

Therefore, researchers have developed a fault model in DOCTOR to emulate processor, memory, and communication faults.

DOCTOR Faults are triggered via time-out, traps, and code modification. When a time-out occurs, the fault injector emulates memory faults by writing over the memory content of the CPU. Software traps trigger non-permanent CPU faults. For permanent CPU faults, software fault injection changes code instructions during compilation to emulate faults that corrupt data or instructions.

DOCTOR consists of five major components: the Experiment Generation Model (EGM), the Experiment Control Module (ECM), the Fault Injection Agent (FIA), the Data Collection Module (DCM), a logging component that collects SWFI data during or after an experiment, and a Data Analysis Module (DAM) that analyzes data collected by the DCM. EGM is responsible for generating workload execution code that contains instructions necessary to carry out processor-fault injections. It also reads user data such as fault type and injection time. In addition, the ECM acts as the controller, by sending commands to the FIA and the DCM. The FIA is responsible for injecting faults and it also controls the execution of the workloads via shared memory and system calls. The DCM's basic function is to continuously log events during experiments.

DOCTOR can emulate both permanent and non-permanent faults. Permanent CPU faults such as data corruption is emulated by changing program instructions during compilation via fault injection. For the simulation of non-permanent faults (i.e., transient or intermittent faults), fault injections issue random faults via traps.

F. FAILURE SIMULATION TOOL (FST)

FST employs wrappers around executable program binaries (e.g., Windows 32 DLL functions) to artificially inject errors or exception calls [1]. A FST interface is instrumented between the SUT executable and the underlying platform's DLL functions. Interactions (i.e., function calls) between the SUT and platform are captured and altered. Functions are modified via the application's Import Address Table (IAT). The IAT keeps track of addresses of imported DLL functions. Thus, by modifying the IAT, testers can re-direct intended function calls to modified functions pointing to the wrapped DLL. The

wrapped DLL will have the ability to call an alternative function, change function parameters, modify function return values, or return an exception or error code.

G. FTAPE

The Fault Tolerance and Performance Evaluator (FTAPE), developed at the University of Illinois, injects faults into CPU memory locations, modules, and disk subsystem. FTAPE injects faults as bit flips to simulate errors. A routine that is executed in the disk system's driver code helps simulate I/O type errors (i.e., bus errors). Fault-injecting drivers added to the operating system create all other errors, hence, reducing the need for modification to the SUT [32].

H. IDEAS FOR SWFI DEVELOPMENT

The following is a list of features from the corresponding SWFI tools that were considered during SWFI development:

- Use of a debugger to step through code during fault injection (XCEPTION).
- Use of fault triggers to invoke fault injection (XCEPTION).
- Use of software traps to emulate faults (FERRARI).
- Use of breakpoints to determine time of fault injection (GOOFI).
- Initiate a time to determine when to inject faults (DOCTOR).
- Emulation of processor, memory, and communication faults (DOCTOR).
- Artificially inject exception calls (FST).
- Processing of bits to emulate errors (FTAPE).

VI. SIMPLE – DESIGN AND IMPLEMENTATION

A. INTRODUCTION

SIMPLE stands for *Software Fault Injection through means of a Mechanized Prototype Lightweight Engine*. It is the authors' attempt to implement a semi-automated, fault-injection test-harness for Java-based systems. The acronym is intended to convey that a Software Fault-Injection process need not be overly complex.¹⁶

Our reasons for developing SIMPLE is two-fold:

Firstly, we sought to provide a software fault-injection tool prototype that facilitates software testing. In case studies described in Chapter VII, we found our tool to be practical and beneficial. For example, SIMPLE exposed bugs in a couple of applications that eluded previous software testing. Ultimately, SIMPLE could serve as a practical resource tool for those interested in learning SWFI.

Secondly, we anticipate our design approaches, implementation choices, and lessons-learned to assist others in the construction of a robust fault-injection software tool. UML diagrams have been provided to supplement our design discussions.

As its name implies, SIMPLE is fairly straightforward to use. Testers configure faults and associated fault attributes via a fault configuration file. The fault attributes determine the fault type, injection time, and source location of a specified fault. SIMPLE then processes these faults and transparently pre-instruments the SUT when necessary. After SIMPLE launches the SUT, faults are injected via fault triggers issued during execution. During this period, testers record suspicious and/or erratic behavior, such as thrown exceptions, performance degradation, and inaccurate program responses.

Our tool focuses only on Java-based systems. In general, Java is becoming the programming language of choice within many defense agencies, especially at SSC-SD.¹⁷

¹⁶ Dr. Jeff Voas himself expressed these exact sentiments in a software fault-injection seminar given at the Naval Postgraduate School (NPS). The seminar, entitled "Discovering Unknown Software Output Modes and Missing System Hazards", was given on April 4, 2002.

¹⁷ Numerous projects at SSC-SD require the use of Java, and many engineers are being retrained as a result. Additionally, SSC-SD is also hiring New Professionals (NPs) with strong Java prerequisites.

Indeed, Java appears to be the front-runner among other languages used to implement and deploy solutions for the US Department of Defense (DoD) [35].

B. ACTIVITY AND CLASS DIAGRAMS

Figure A-1 in Appendix A depicts a UML activity diagram for SIMPLE. More specifically, SIMPLE engages in the following activities: *Read-Faults*, *Instrument-SUT*, *Deploy-Faults*, *Execute-SUT*, *Trigger-Faults*, and *Inject-Faults*. The *Trigger-Faults* and *Inject-Faults* activities are separate because a fault is not always injected when triggered. For example, during the fault-deployment stage, each fault is mapped to a *fault-location*, which consists of a specified class and a line number within that class. Whenever execution passes through a fault-location, the corresponding fault will always be “triggered.” The actual injection of that fault, however, depends on the current values of its associated fault attributes. For instance, some fault attributes, such as *fault probability*, may cause the fault to be suppressed. Fault attributes are discussed in later in this chapter.

Figures A-2 and A-3 depict UML class diagrams that illustrate the basic constituents of the SIMPLE architecture. The principal components include a *SimpleHarness*, a *FaultParser*, an *EventThread*, a *FaultManager*, an *SUTInstrumentor*, and a *Fault* object.

The main program for SIMPLE is the *SimpleHarness* component. It encompasses all the activities described in the previous activity diagram. The arguments to *SimpleHarness* specify the application name and supporting classpath elements. This provides the SIMPLE components with information about the SUT. The *SUTInstrumentor*, for instance, uses the provided classpath information to locate SUT classes for instrumentation.

The *FaultParser* component reads and parses fault information from the fault configuration file. Depending on the particular type of fault, the *FaultParser* either:

- 1) Constructs a *Fault* entity to add to the *FaultManager*, or
- 2) Invokes the *SUTInstrumentor* to pre-instrument faults into the SUT.

Faults that require pre-instrumentation include memory-exhaustion, processor-exhaustion, forced-delays, and exception-throwing faults. These faults are discussed in the next section.

The *FaultManager* component manages faults during SUT execution. When a fault trigger is encountered, the *FaultManager* is preempted to process all corresponding faults that apply specifically to the trigger. The *FaultManager* then checks these faults against their fault attributes to determine whether they are to be injected.

The *EventThread* (also can be referred to as the *FaultInjector*) component monitors execution and issues fault triggers based on user-specified fault locations pre-configured into SIMPLE. Once a fault trigger is issued, the SUT execution pauses while the *FaultManager* processes faults and considers them for possible injection into the SUT. Afterwards, the SUT resumes execution until the next fault trigger occurs.

The *Fault* component defines a fault specification in SIMPLE. In particular, each fault shall contain user-specified information indicating the following:

- 1) *Where* it will be injected,
- 2) *When* it will be injected,
- 3) *How* it will be injected, and
- 4) *What* will be injected.

Currently, there are two categories of fault types: Those that are pre-instrumented into the SUT, and those that are not. The next session discusses the different types of SIMPLE faults.

To control pre-instrumented faults embedded within the SUT, the *EventThread* component communicates with the *SimpleHelper* component that acts as an auxiliary control component. Its primary role is to properly regulate fault activation on the SUT. However, the caveat here is that the *SimpleHelper* class must be integrated into the SUT.

C. FAULTS IN SIMPLE

SIMPLE is primarily a state-based fault-injection engine. That is, it has been designed to mutate internal data variables within the SUT. Additionally, SIMPLE can be

classified as a glass-box fault-injection technique since it taps into the inner workings of the software. As it pertains to faults, we implement fault models that emulate internal state corruption. The *Fault* subclasses defined in the class diagram provide the hierarchical infrastructure that enables SIMPLE to inject state-based faults.

Recall that faults are entered into SIMPLE via a fault configuration file. Therefore, testers must properly define fault location, fault type, and other fault attribute information for each fault via XML¹⁸ notation. Consequently, it is possible for many faults to exist at a single class location. SIMPLE will issue an error message when an invalid fault input is encountered in the fault configuration file. Figure 1 below shows a sample fault input listed in a configuration file.

```

12      <!-- Test Case 1: withinRangeNetworkParameters-->
13      <Fault class="csma.app.NetworkSimulationMainTest" lineNo="58" numOfInvoc="1">
14          <PrimField varName="numOfRuns" valToSet="-999"/>
15      </Fault>
16
17      <!-- Test Case 2: outOfRangeNetworkParameters-->
18      <Fault class="csma.app.NetworkSimulationMainTest" lineNo="92" numOfInvoc="1">
19          <PrimField varName="packetLength" valToSet="-999"/>
20      </Fault>
21
22      <!-- Test Case 3: queueSingleElementToQueue -->
23      <Fault class="csma.app.NetworkSimulationMainTest" lineNo="128" numOfInvoc="1">
24          <PrimField varName="maxPackets" valToSet="-999"/>
25      </Fault>
26
27      <!-- Test Case 1: queueSingleElementToQueue -->
28      <Fault class="csma.client.PacketQueueTest" lineNo="44" numOfInvoc="1">
29          <ObjLocal varName="packet" setToNull="true" />
30      </Fault>
31

```

Figure 1. Sample Fault Input

Refer to Appendix B for a complete grammar specification of the fault configuration file.

1. Fault Types

The fault types SIMPLE currently supports are *variable-mutation*, *memory-exhaustion*, *processor-exhaustion*, *thrown-exception*, and *forced-delay* faults. All faults, except the variable-mutation fault, require pre-instrumentation into the SUT. That is, these faults must be physically embedded into the byte-code of the SUT. Sections D and F of this chapter describes how fault pre-instrumentation occurs in SIMPLE.

¹⁸ As developed by the World Wide Web Consortium (W3C), the Extensible Markup Language (XML) is the universal format for structured documents and data on the Web. More info can be found at [36].

Variable-mutation faults model corrupted state variables presumably caused by race conditions, class misuse, or incorrect logic. These faults can be applied to practically any class field or local variable within the SUT, regardless of scope or visibility.

Memory-exhaustion faults simulate both memory leaks and reckless memory consumption within the SUT. When injected, these faults instantiate a number of arbitrary object instances into the memory heap of the SUT. The number of objects instantiated is configurable in the fault configuration file.

Processor-exhaustion faults create a number of executing threads within the SUT. It exhausts CPU resources used by the SUT. The number of thread processes created is configurable in the fault configuration file.

Thrown-exception faults invoke exceptions at user-specified locations within the SUT. These faults are particularly useful in assessing the fault-handling and error-recovery mechanisms of the SUT. The type of exception is configurable in the fault configuration file.

Forced-delay faults cause delays to occur at user-specified locations within the SUT. These faults force timing errors and also mimic time-consuming tasks. The length of the delay is configurable in the fault configuration file.

2. Fault Attributes

In general, fault attributes identify the *what*, *when*, *where*, and *how* properties for a fault in SIMPLE. For example, the *what* determines the type of fault to be injected. The *when* specifies the fault activity time-frame. The *where* indicates the location of the fault within the SUT. The *how* specifies how the fault is to be injected. Currently, SIMPLE supports the following fault attributes:

The *fault-type* attribute specifies the kind of fault to be injected. This attribute determines whether pre-instrumentation is necessary for the specified fault. Except for variable mutation faults, all other faults require pre-instrumentation.

The *class-name* attribute determines the name of the SUT class in which the fault will reside. The tester must specify the fully qualified class name in the fault configuration file.

The *line-number* attribute indicates the line number of the class where the fault will be located. Only certain line numbers can be selected within the source code.

The *enable* attribute determines whether the fault is active or inactive during fault-injection. This attribute takes priority over all other attributes that also specify a fault activation status.

The *start-time* attribute defines the beginning time that the fault will become active. The default value for this attribute is -1 , which means that the fault is active at the onset.

The *end-time* attribute denotes the finish time that the fault becomes inactive. The default value for this attribute is -1 , which means that the fault is active indefinitely.

The *activateAt* attribute specifies a location within the SUT that the fault is to be activated when encountered during execution.

The *deactivateAt* attribute indicates the location within the SUT that the fault is to be deactivated when encountered during execution.

The *probability* attribute describes the injection probability of the fault. Probability values range from 0.0 to 1.0. The default value for this attribute is 1.0, which means that the fault is always active when triggered.

The *number-of-invocations* attribute determines the number of times a fault is injected. The default value for this attribute is -1 , which means that the fault is always active when triggered, as far as this attribute is concerned. Of course, other fault attributes may also determine whether the fault is active.

The *variable-name* attribute identifies the target program variable to be mutated when the fault is injected. The variable can either be a class field member variable or a local variable.

The *set-to-value* attribute indicates the *value* to be applied to the target variable. The default value for this attribute is *random*, which means that a random value is applied to the type-specific variable.

The *set-to-null* attribute determines that a *null-value* will be applied to the target variable. The default value for this attribute is *false*, which indicates that a *null-value* is not applied to the target variable. This attribute applies only to variables that are instantiations of object classes. For this reason, primitive variables to be corrupted, such as integers, cannot use this attribute.

The *arg* attribute is a general-purpose attribute used that helps deploy memory-exhaustion, processor-exhaustion, thrown-exception, and forced-delay faults. For example, the memory-exhaustion fault uses *arg* to determine the number of objects to instantiate within the SUT.

3. Fault Triggers

Each fault location¹⁹ specified in the fault configuration file is associated with a corresponding *fault trigger* prepared during the fault-deployment phase. The fault trigger represents a specialized run-time event used to invoke the fault-insertion process. When a fault location is encountered during execution, a fault trigger prompts SIMPLE to inject all applicable faults that correspond to the fault location. However, faults can be triggered, but not necessarily injected (i.e., activated). This solely depends on the current values of their fault attributes. Any single fault attribute can suppress a fault from being injected, regardless of the values of other attributes. For example, if the *number-of-invocations* attribute is 0, then the triggered fault will not be injected. Similarly, a *probability* attribute of 0 will also suppress the triggered fault from being injected.

D. JAVA TECHNOLOGIES

This section describes some of the Java Technologies and APIs that were explored for SIMPLE. It also describes the technologies ultimately selected and utilized for SIMPLE development. Of particular interest is the discussion in Section F of this chapter concerning pre-instrumentation issues.

¹⁹ Fault location refers to the location defined by a fault's class-name and line number attributes.

1. Candidate Technologies

During the early development stages, we considered using a *metaobject* protocol as an approach to inject mutation faults into a Java application. In a metaobject protocol, a *base-level object* and a *meta-level object* are both bound to classes during one of the following stages: compile-time, load-time, or run-time. The base-level object exposes behavioral events invoked by class instances during execution. Base-level behavioral events typically include *method calls*, *exception invocations*, and *field-data access*. The meta-level object intercepts these events before reaching the application for further processing [14]. For example, depending on the application utilizing this protocol, an intercepted event can be processed in one of the following ways:

- 1) The event can be passed on to the application,
- 2) The event can be suppressed from reaching the application, or
- 3) The event can be modified upon arrival, and then passed on to the application.

Thus, this protocol provides for a powerful mechanism for implementing software fault-injection. [13] provides a summarized comparison matrix of reflective Java API that can be utilized to implement a metaobject protocol.

Javassist and AspectJ were investigated early in this study. Both technologies provide a high-level, feature-rich Java API, and are distributed under an open-source license. However, they differ in their implementation of the metaobject protocol.

Javassist stands for *Java Programming Assistant* and was developed at the Tokyo Institute of Technology [37]. It uses load-time reflection for modifying Java byte-code and defining new class elements. In addition, it consists of a Reflection API that provides metaobject control over participating application classes.

By using Javassist, metaobject bindings and class redefinitions can be performed at load-time. To accomplish this, a specialized classloader invoked at load-time intercepts and modifies class byte-definitions accordingly. Unless saved to the disk, class redefinitions are only active during the execution of the application. A Javassist program

harness must launch the application in order to utilize the classloader. One caveat of Javassist is that it cannot modify Java system classes at load-time due to Java security.²⁰

```
1 import javassist.*;
2 import javassist.reflect.*;
3
4 public class VerboseMetaobject extends Metaobject {
5
6     // Intercepts constructor calls
7     public VerboseMetaobject(Object self, Object[] args) {
8         super(self, args);
9         System.out.println("*** constructed: " + self.getClass().getName());
10    }
11
12    // Intercepts field reads
13    public Object trapFieldRead(String name) {
14        System.out.println("*** field read: " + name);
15        return super.trapFieldRead(name);
16    }
17
18    // Intercepts field writes
19    public void trapFieldWrite(String name, Object value) {
20        System.out.println("*** field write: " + name);
21        super.trapFieldWrite(name, value);
22    }
23
24    // Intercepts method calls
25    public Object trapMethodcall(int identifier, Object[] args)
26        throws Throwable
27    {
28        String methodName = getMethodName(identifier);
29        System.out.println("*** trap: " + methodName
30            + "() in " + getClassMetaobject().getName());
31        return super.trapMethodcall(identifier, args);
32    }
33 }
```

Figure 2. VerboseMetaobject class

Figure 2 shows source code for a `VerboseMetaobject` class that subclasses the `metaobject` class defined in Javassist. The `trapFieldRead` method (lines 12-16) and `trapFieldWrite` (lines 18-22) method intercept class field variables accessed during execution. In other words, the `trapFieldRead` and `trapFieldWrite` methods are triggered each time program variables are accessed and modified during execution, respectively. Similarly, the `trapMethodCall` method (lines 24-32) intercepts class methods called during execution. The method arguments provide details describing the program variables and methods being intercepted during run-time. For example, on line 13, the `name` parameter of the `trapFieldRead` method identifies the actual variable being accessed. Similarly, the `identifier` and `args` parameters of the `trapMethodCall` method, shown in line 25, identify the application method being invoked.

²⁰ Java system classes include specialized classes that begin with `java.*` or `javax.*`. Fortunately, Javassist provides additional tools to modify these classes at compile-time if so desired.

AspectJ, distributed by Xerox PARC, is a Java extension that supports the aspect-oriented programming (AOP) paradigm [38]. Aspect-oriented software development (AOSD) supports the use of separation of concerns (SOC) in software development. The techniques of AOSD make it possible to modularize crosscutting aspects of a componentized system [64]. Keeping in tune with the AOP programming model, AspectJ can programmatically modularize crosscutting concerns that inherently exist in many software implementations. For example, logic scattered throughout the source code can be made accessible to the programmer and used in a modularized programming construct. Therefore, extended application functionality can be constructed in a manner such that it does not require modification of existing infrastructure. Specifically, *joinpoint* and *pointcut* programming constructs enable programmers to capture events at well-defined areas of interest (i.e., clearly defined juncture points) in a program's execution. Examples of well-defined areas of interest include method calls and data-field access. Additionally, these programming constructs derive *advice* methods that encompass additional business logic that are applied at corresponding juncture points.

```

1 import java.lang.NullPointerException;
2
3 privileged aspect ExampleAdvice
4 {
5     private static final int NEGLECT = 0;
6     private static final int DUPLICATE = 1;
7     private static final int THROW_EXCEPTION = 2;
8
9     // Assume that controlFlag can be externally manipulated
10    public static int controlFlag = -1;
11
12    void around(CollectionObject collection, Object obj): target(collection)
13        && args(obj) && execution(* CollectionObject.add(Object))
14    {
15        if (controlFlag == NEGLECT)
16        {
17            return; // Ignore method call
18        }
19        else if (controlFlag == DUPLICATE)
20        {
21            proceed(collection, obj); // Add duplicate to vector
22        }
23        else if (controlFlag == THROW_EXCEPTION)
24        {
25            throw new NullPointerException("Forced Exception"); // Throw exception
26        }
27        proceed(collection, obj); // default action ...
28    }
29 }
30 }

```

Figure 3. Example Advice

As an illustration, Figure 3 shows sample source code for an *advice* that affects the behavior of the *add* method of a *CollectionObject* class (alluded to in line 13) whenever called. Depending on the current value of the control flag, one of the following behaviors will occur:

- 1) The method will neglect to add the object,
- 2) The method will add multiple objects, or
- 3) The method will throw a *NullPointerException* back to the caller.

Note that the *advice* implementation operates as a wrapper to the *add* method.

AspectJ requires a full recompilation of application source code in order to integrate *advice* constructs. Moreover, this recompilation requires use of an AspectJ compiler.

Due to the abilities of capturing and changing software behavior, both Javassist and AspectJ can be utilized for SWFI purposes. We considered both of them for the design of SIMPLE. Javassist and AspectJ can be especially powerful in assessing system interface interactions at the unit-, component-, or subsystem-level. Binder advocates integration testing to search for faulty components that lead to inter-component failures [27]. These two technologies can help facilitate this and other forms of similar testing strategies.

One problem, however, is the issue of *fault-injection granularity*. This refers to *where* a particular fault can be injected within the application. In the metaobject protocol, the only events that can be intercepted are at well-defined junctures within the application. These junctures include method calls, constructor calls, field-data access, exception invocations, and a few others. In this case, faults can only be applied at these junctures. In Figure 3, we can see that faults were injected at each immediate *add* method invocation; however, it is not possible to arbitrarily specify a line number within the body of the *add* method to inject our fault.

Being able to specify a flexible location to inject a fault is an especially important feature we wanted SIMPLE to address. Arguably, many software failures can be attributed to inter- and intra-component interfaces. However, it would be interesting to

investigate software failures caused by faults located in non-obvious areas of the code. By non-obvious, we mean code not located at well-defined junctures. Whether this strategy is worthwhile to pursue remains a topic of future research. In the next section, the following technologies were used to accomplish this goal. Problems encountered are discussed later in Section F.

2. Selected Technologies

The three major challenges faced when designing SIMPLE were:

- 1) How to define fine-grained injection points within an application?
- 2) What kinds of faults can be injected in the application?
- 3) How can exception and similarly related faults be injected?

The first question was briefly touched on in the previous section. In this case, we want to somehow extend the *fault-injection coverage* of an application to give testers more flexibility in *where* they can locate a particular fault, not restricting them to placing faults at well-defined junctures.

The second question is concerned with the variations of fault types that SIMPLE could allow. In our study, we chose to emulate real-world faults as much as possible. Hence, to do this, we focused on employing two well-documented fault-injection techniques: data- and code-based mutations [3]. If implemented correctly, these two techniques could support a wide range of fault models to be emulated.

The third challenge was to figure out how exception faults could be emulated. Emulating exception faults would be an important feature in SIMPLE, especially for the practice of *fault acceleration*. Fault acceleration refers to the process of accelerating an application's failure rate in a controlled environment within a particular time frame [7]. (The concept of fault acceleration is revisited in Chapter VII.) By simulating timely exception faults, for instance, we would be able to measure the fault-tolerant characteristics of the SUT. Of course, SIMPLE could easily corrupt the program to normally invoke the desired exception. In this case, we would have to wait until these faults fully manifested themselves during execution. However, it would be more efficient to programmatically throw the exception at any desired location. For example, a feasible

approach was illustrated in Figures 2 and 3 using the metaobject protocol approach discussed earlier. Fortunately, the Java Platform Debugger Architecture (JPDA) and Compaq JTrek APIs adequately address this issue. In fact, these technologies proved to be a much elegant solution than the metaobject protocols previously mentioned.

Sun Microsystems' JPDA provides [39] provides debugger support for the Java 2 Platform, along with defining high- and low-level standardized debug interfaces. The JPDA comes with a complete reference implementation that is publicly available for download. In particular, the JPDA offers an API equipped with debugging features such as *breakpoint* processing, code stepping, variable evaluation and modification, and *watchpoint* configuration. These features easily gather insight about an application running in a targeted Java Virtual Machine (JVM). Moreover, the JPDA launches the SUT in a separate debug JVM process. In this regard, the JPDA back-end becomes a remote test harness to the SUT. In turn, the debug JVM provides hooks that allow a JPDA back-end to access run-time information.

When considering possible designs for SIMPLE, the JPDA offered several highly desirable, crucial services. First of all, the JPDA supports the ability to configure breakpoints practically anywhere in the program, ensuring that the fault-injection coverage is expanded to include “hard-to-reach” areas of the code. Secondly, the JPDA supports accessing and modifying data variables within the debuggee application at run-time. Hence, it provided a mechanism for mutating data variables, thus affecting application state. As a result, SIMPLE could conceivably emulate faults caused by race conditions, misuse of class methods, or incorrect programmer logic.²¹ Lastly, the JPDA technology provides a programmatic approach to automate the fault-injection process without intervention by the tester. Binder promotes the use of automatic instrumentation since manual instrumentation is error-prone and time-consuming [27].

With the granularity and fault type issues resolved by JPDA, we focused our attention to addressing the third question, which concerned emulating exception faults. Unfortunately, the JPDA did not provide a facility to inject exception invocations. As a workaround, JPDA could be used to indirectly force exceptions by mutating data

²¹ We rationalized that code-mutation resulted in data-mutation. Hence, we felt we were not obligated to implement this type of mutation scheme.

variables at appropriate times. For example, consider the code shown in Figure 4 below. One possible way to invoke a *NullPointerException*, say, on line 637, would be to mutate the *result* variable to a *null* value.

```
624    ResultSet result = executeQuery("SELECT * FROM FLIGHT;");
625
626    try
627    {
628        while (result.next())
629        {
630            if (vector == null)
631            {
632                vector = new FlightVector();
633            }
634
635            flightNumber = (String) result.getString("FLIGHT_NUMBER");
636            departDate = (String) result.getString("DEPART_DATE");
637            departTime = (String) result.getString("DEPART_TIME");
638            departCity = (String) result.getString("DEPART_CITY");
639            arriveDate = (String) result.getString("ARRIVE_DATE");
640            arriveTime = (String) result.getString("ARRIVE_TIME");
```

Figure 4. Arbitrary Code Segment

However, how could other exception types be thrown? What combinations of variables would have to be mutated to incur an *OutOfMemoryException*? Is it even possible to do this by only considering variables? How long would it take? In spite of these issues, we abandoned implementing exception faults in JPDA. Instead, a pre-instrumentation approach was considered as a possible solution. Even though this meant physically changing Java class byte-code, we believed this to be our only viable alternative.²² Fortunately, various open-source Java byte-instrumentation tools were available that allowed for this type of experimentation. Therefore, developing a byte-code instrumentation tool became a non-issue.

The Compaq JTrek Technology [40] provides an API to analyze, modify, and profile activity of Java class files. By scanning class byte-code, JTrek formally breaks down Java code into a hierarchy of analytical components: class files, fields, methods, statements, local variables, and instructions. Each hierarchical component comes with its own set of API for acquiring more detailed information about itself. More importantly, JTrek comes equipped with byte-instrumentation API for inserting statements and

²² Both the FERRARI and DOCTOR fault-injection tools utilize a pre-instrumentation approach where software trap instructions are inserted into byte-code. This was done specifically to simulate processor faults.

instructions into Java classes. This key capability, along with its class-scan capability, influenced us to consider using this API for SIMPLE.

JTrek offers the capability to insert a method call into a Java class. Thus, this afforded us a pre-instrumentation capability we could integrate into SIMPLE. In addition, this pre-instrumentation would only affect precompiled byte-code, and not source code.²³ Given this, we were now able to implement auxiliary methods that could throw a diverse set of exceptions. These specialized methods would, in turn, be automatically pre-instrumented into a Java class when called upon. This proved to be a very effective technique for throwing exceptions. It gave us the ability to inject other types of faults, such as memory-exhaustion, processor-exhaustion, and forced-delay faults. The only caveat is that pre-instrumentation must be done before the SUT application is launched.

E. STATE AND SEQUENCE DIAGRAMS

Figure A-4 in Appendix A illustrates a state diagram depicting a detailed account of the activities previously shown in the activity diagram. It incorporates processes that are embodied by the JPDA, JTrek, and Xerces APIs. For example, the JPDA API plays a significant role in many of the diagram states such as SUT EXECUTE, RESOLVE FAULT, TRIGGER FAULT, and INJECT FAULT. On the same token, the JTrek API is largely responsible for the INSTRUMENT SUT state. The Xerces API [41] is responsible for the READ FAULT state. Since the fault configuration file uses an XML specification, Xerces was used to parse the XML nodes of this file. See Appendix B for the grammar specification of the fault configuration file.

The RESOLVE FAULT and TRIGGER FAULT states specifically handle JPDA debugger events generated by the target JVM. The purpose of these states will become more apparent in the sequence diagrams to follow. The START TIMER state is discussed in Section F.

The RESOLVE FAULT state encompasses the process of configuring breakpoints (i.e., fault triggers) within the target JVM. A breakpoint is set for each fault specified in

²³ This was a requirement we wanted to strongly adhere to as much as possible.

the fault configuration file. Recall that fault attributes provide class name and fault line number information in the SUT. This fault-to-breakpoint mapping occurs during run-time on a per class basis as classes are being loaded into the target JVM. Since specific debugger events notify loading of a particular class, SIMPLE can programmatically resolve any faults (i.e., configure any breakpoints) that are associated with the currently loaded class. Resolving faults basically means the process of configuring appropriate breakpoints for each fault.

The TRIGGER FAULT state encompasses actions taken by SIMPLE when a breakpoint (i.e., a fault trigger) is encountered in the target JVM. When this occurs, the target JVM pauses and SIMPLE delegates control to the *FaultManager*. In turn, the *FaultManager* processes all faults that apply to that particular fault trigger. (At this point, this activity is encompassed within the INJECT FAULT state.) The target JVM resumes after the *FaultManager* has processed all faults.

Figure A-5 depicts a UML sequence diagram illustrating the top-level interactions between the *SimpleHarness*, *EventThread*, and *Target JVM*. It presents another high-level view of the activity diagram.

In Figure A-6, a UML sequence diagram illustrates how SIMPLE reads and parses faults into the system. Depending upon the type of fault being parsed by the *FaultParser*, faults are either added to the *FaultManager* or pre-instrumented within the SUT. Note the methods that are called upon to set the fault attributes of the *Fault* object. Refer to the psuedo-code portion of the diagram for an algorithmic view of this procedure.

The UML sequence diagram in Figure A-7 demonstrates how SIMPLE resolves faults during execution. It presents another view of the RESOLVE FAULT state in the state diagram. When a class is loaded, the target JVM immediately notifies the *EventThread*. The *EventThread* then extracts the class from the debugger event, searches the *FaultManager* for all correlated faults, and sets breakpoints based on fault-location. Refer to the psuedo-code portion of the diagram for an algorithmic view of this procedure.

In Figure A-8, a UML sequence diagram illustrates how SIMPLE processes fault triggers and faults during execution. It presents another view of the TRIGGER FAULT and INJECT FAULT states in the state diagram. When a breakpoint is encountered, the target JVM immediately notifies the *EventThread*. The *EventThread* then extracts class and location information from the debugger event. Next, the *FaultManager* receives the extracted class information and processes the faults. In Figure A-8, note that a call to the *isTimeToInject* method determines whether the fault is injected or not. More precisely, this method specifically examines the current values of the fault attributes. Refer to the psuedo-code portion of the diagram for an algorithmic view of this procedure.

F. ISSUES, CAVEATS, LIMITATIONS, AND LESSONS-LEARNED

This section describes some existing problems encountered, plus some caveats and limitations associated with using SIMPLE. We also describe the rationale behind some of our design decisions.

1. Java Virtual Machine (JVM) Compatibility

SIMPLE is only compatible with Java-2 compliant JVMs that implement the Java Debugger Interface (JDI) [39].

2. Requires Compilation with Debug Option

SIMPLE heavily relies on debugging information specially generated via the `-g` debug option of the *javac* compile tool. In particular, SIMPLE uses line number and local-variable debug information for identifying fault locations and candidate corruption variables, respectively. For this reason, the SUT must be recompiled accordingly to ensure that SIMPLE functions properly. A side-effect is that the generated Java class files will be larger due to the extra-embedded debug information.

3. How to Handle Fault-Injection Time?

As discussed earlier in Section C, the *Fault* class introduces time-related attributes that specify the activation time for a particular fault object. An interesting problem, however, involves the handling of *fault-injection time*. This refers to time observed during fault-injection testing.

Time issues arise because of the following reasons:

- 1) SIMPLE is essentially a remote application to the SUT.

- 2) Faults are maintained by mechanisms residing on both SIMPLE and the SUT.
- 3) Fault-injection time does not always begin at the exact moment that the SUT is launched.

Since fault mechanisms reside in both SIMPLE (for variable-corruption faults) and the SUT (for pre-instrumented faults), it is ideal that each application obtains time information from the same source. Keep in mind that SIMPLE and the SUT are separate applications executing in different JVMs. If both applications were to keep track of time independently, then an inherent drift incurred by each application would pose a problem. For example, the resulting loss of *time-synchronicity* would cause time-activated faults in both applications to be fired at unexpected times during execution. Hence, a protocol needs to be implemented in order to maintain clock synchronization between SIMPLE and the application.

In the current implementation, SIMPLE maintains an internal timer that provides fault-injection time to both SIMPLE and the SUT. (Future implementations may consider adding an external timer server to serve both applications.) Unfortunately, overhead is created since SIMPLE has to continuously update the SUT with fault-injection time. Depending upon the time update rate, this may or may not be a problem during testing. In this thesis, we did not explore the implications caused by *time-communicated* overhead. Therefore, we defer to future work in SIMPLE to assess this.

Determining the start of fault-injection time posed another challenge. In early versions, SIMPLE activated its internal timer immediately after the SUT was launched in the target JVM. However, we encountered problems. In one of our experiments, we specified a fault in the fault configuration file to be activated within some predetermined time frame. After SUT was launched, we realized that the SUT did not start until a button was pressed on the SUT GUI. As a result, the SUT would remain idle, but SIMPLE's internal timer incorrectly continued to count. Eventually the activation time frame defined by the fault expired even though the SUT never was actually started. Figure 5, for example, shows an application GUI used in one of our case studies, where the application did not start until the "Run" button was pressed.

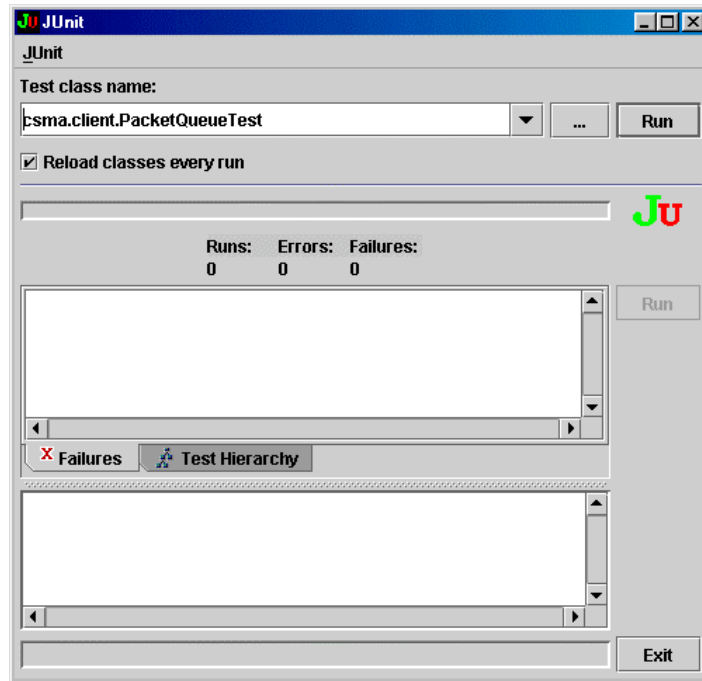


Figure 5. Application that is Activated by the RUN Button

To solve this problem, we incorporated a scheme in which the SUT notifies SIMPLE to start or reset its internal clock timer. Fortunately, we were able to use existing SIMPLE infrastructure to implement this. For instance, in the same manner that fault breakpoints were configured, specialized timer breakpoints could also be set to invoke a *timer-reset* action in SIMPLE. Thus, we designed it so that the tester simply specifies the precise location for this *timer-reset* action via the fault configuration file.

More technically, a *StartTimeEvent* class was added to SIMPLE that encompassed this timer-reset action. The *StartTimeEvent* class was made a subclass of the *Fault* class so that *StartTimeEvent* objects could be added to the *FaultManager* component without loss of generality. When a breakpoint occurs, the *FaultManager* processes *StartTimeEvent* objects just as it would with *Fault* objects. If the breakpoint correlates to the *StartTimeEvent* object, then the *EventThread* component automatically resets its internal clock timer. This is the key activity for handling fault-injection time. Hence, a button press in Figure 5 would cause a *StartTimeEvent* object to be processed. This, in turn, would reset SIMPLE's internal clock timer. The START TIMER state shown in Figure A-4 in Appendix A diagrammatically depicts this reset action.

4. Pre-instrumentation – A Necessary Evil

Pre-instrumentation is a necessary evil for SIMPLE, especially for the emulation of exception faults. Besides modifying the source code, to our knowledge no practical means exist to remotely invoke exceptions into the various JVM threads. Recall that SIMPLE launches the SUT in a separate process (i.e., in a separate debug JVM). The only viable approach was to physically insert method calls to an integrated client-side component (i.e., the *SimpleHelper* component). This technique would natively invoke exceptions within the host JVM.²⁴ Fortunately, these insertions would be transparently made to the Java class byte-code, and not the actual source code. As described in Section D, the Compaq JTrek API automated this pre-instrumentation procedure.

Pre-instrumentation, however, runs the risk of accidental software deployment. In this case, software production code should not contain active debugging statements, or other live macros that were specifically used for testing.²⁵ Binder [27] advocates that instrumentation be done on a copy of the class under test. Then the instrumented class is discarded or archived after testing has been completed. To account for this, SIMPLE instruments a copy of the Java class byte-code file, not the original. SIMPLE then strategically places this copy in its classpath for proper usage during fault-injection testing. In this way, SIMPLE restricts the mutated class for only its own use.

5. Pre-instrumentation Behavior is Inconsistent

SIMPLE locates an *executable-statement* by matching the line number specified in the byte-code definition to the line number specified in the fault.²⁶ In the event of a match, SIMPLE inserts appropriate byte-code logic into the class definition at the appropriate location. If a statement is not found, then SIMPLE issues an error message indicating that the fault could not be deployed. Figure A-9 depicts SIMPLE's pre-instrumentation algorithm.

The matching algorithm is relatively straightforward to implement. It inserts fault-code into a location specified by the tester. However, the algorithm implemented via

²⁴ We had to relax a personal requirement prohibiting the modification of the SUT.

²⁵ These should not necessarily be removed, but they definitely should be disabled for deployment.

²⁶ An executable-statement is a single line of Java code that is executed during execution.

SIMPLE falters occasionally due to a limitation found in the JTrek API. The problem is that SIMPLE occasionally inserts byte-code a few lines off from the original designated location. In some cases, SIMPLE skips insertion of a byte-code even when required to do so. Obviously, this anomalous behavior causes incorrect software assessments by the tester during fault-injection testing. After further investigation, this problem originates from the fact that line number information is not given for *declarative* Java statements, such as try-catch statements. As a result, these declarative statements are ignored entirely by the JTrek API. In turn, this throws off the statement line number count that JTrek maintains on the class' behalf.²⁷

SIMPLE compensates for this problem by utilizing a workaround fix. Specifically, this workaround scheme utilizes both source code and byte code of the specified class to determine the location of a statement. When encountering a declarative statement during the statement parse, SIMPLE refers back to the source code to determine the statement's exact line number within the class. SIMPLE then uses this information to resynchronize the line number count for the class. This is an unfortunate problem, but we intend to fix it in our next release.

6. Requires Source Code and Strong Familiarity Thereof

During fault-injection testing, SIMPLE requires testers to have full access to the SUT source code for the following reasons:

- 1) Testers must ensure that the SUT is recompiled with the javac debug option turned on. In addition, the *SimpleHelper* component must be recompiled along with the SUT. SIMPLE does not currently automate SUT recompilations.
- 2) In order to appropriately specify faults in SIMPLE, a class-name and line number must be supplied as fault-attributes. Hence, the tester must be able to freely examine (or update) the source code to determine where to place faults.
- 3) Testers should be familiar with the inner workings of the software in order to understand software responses to triggered fault injections. The source code,

²⁷ This is not a bug in JTrek since it was documented to behave in this manner. It is just that SIMPLE is attempting to use the API in a manner that was not intended.

along with other available resources at their disposal, can supply a valuable point of reference. Examples of other available resources include developers, design documents, manuals, and previous test materials. In addition, source code familiarity can help testers focus on perceived trouble spots in the software. Therefore, testers can target faults more productively and efficiently.

- 4) To compensate for the pre-instrumentation flaw in SIMPLE described earlier, the pre-instrumentation process will parse both the byte code and the actual source code of the class in question; this is another reason that source code is required during fault-injection testing. If source code is not provided, then faults incorrectly may be instrumented at ambiguous locations within class files. Consequently, this may have an adverse effect on software fault-injection post-analysis. As mentioned before, we are currently investigating alternative work-around solutions that do not rely on contrived processes that algorithmically analyze source code.

7. Heisenberg’s Uncertainty Principle

As applied to software, *Heisenberg’s Uncertainty Principle* states that the process of observing one aspect of software can introduce artifacts that adversely alter what is ultimately being observed [42]. Unfortunately, SIMPLE adheres to this principle due to the amount of overhead during fault injection.

The following scenario describes the overhead SIMPLE incurs during a typical fault-injection session:

The debug JVM encounters a *Breakpoint* event and sends a *Fault-Trigger* message to SIMPLE. SIMPLE receives this message and processes predefined injection faults. For each fault that satisfies current fault-injection criteria, SIMPLE sends an appropriate *Variable-Change-Request* message to the debug JVM. The debug JVM receives the *Variable-Change-Request* message and processes it accordingly, thus changing internal state variables as needed.

The above scenario demonstrates how SIMPLE uses a remote debugging approach toward fault-injection. While intrusive in nature, SIMPLE can remotely assess and affect SUT behavior via a communications link established with the debug JVM.

This communications link provides a dedicated channel for transmittal of debug messages between the debugger and debuggee.

Though remote debugging has many advantages, some inherent disadvantages include that of overhead. In [43], overhead is defined as the “... use of computer resources for performing a specific feature.” In our case, we use the term as it relates to Software Testing.

Unfortunately, overhead is unavoidable and inevitable in software testing. Thus, it is sometimes necessary to integrate monitoring mechanisms to evaluate internalized behavior during execution. Consequently, the software may have to be modified (i.e., made more testable) to accommodate for this. An unfortunate effect is that these modifications potentially can affect software timing and performance behavior. This can be especially problematic if not accounted for properly during testing, causing post-analysis to yield misleading results.

In SIMPLE, one reason for the incurred overhead is due to the constant traffic of debug information being maintained between the debugger and debuggee. For instance, the debug JVM sends a *Fault-Trigger* message in response to an encountered *Breakpoint* event. After processing a *Fault-Trigger* event, SIMPLE sends a *Variable-Change-Request* back to the debug JVM. All of this interfacing takes valuable time. Thus, the frequency of processed events and transmitted messages influences overhead and ultimately, the overall performance of the system.

```
27
28     int i = 0;
29     int factor = 2;
30     for (int j = 0; j < 1000; j++)
31     {
32         i = j * factor;
33         factor = factor * 2;
34         System.out.println(i + ", " + j + ", " + factor);
35     }
36
```

Figure 6. Code Snippet

To illustrate this, consider the code snippet shown in Figure 6 that depicts a tightly wound *for*-loop. If a variable-mutation fault were inserted at line 33, then a drastic decrease in performance would occur. This is caused by the ensuing fault being

triggered, evaluated, and injected at each iteration of the loop. As a consequence, this considerable lag conceivably can cause other aspects of the software to fail, or race conditions to occur.²⁸ In this case, the fault did evoke an incorrect software response, but this may not be what the tester had originally intended for the fault. This would be unacceptable when applying fault-injection testing to real-time systems.

Throughout its development, we attempted to minimize SIMPLE overhead as much as possible so that it did not directly or indirectly invoke unintended software faults during a fault-injection session. Thus, this prompted us to limit some of the features we desired for SIMPLE. For example, the JPDA allows for the registration of other debug events, such as *Step*, *Method-Entry*, *Method-Exit*, *Exception-Trigger* and *Watchpoint* events.²⁹ Conceivably, many of these debug events can be extremely useful to SIMPLE for obtaining profile information concerning the SUT. The *Step* Event, in particular, would be especially useful in measuring dynamic execution coverage metric. Unfortunately, due to the potential extra overhead associated with these features, a critical design decision was made to just use *Breakpoint* events in SIMPLE.

To compensate for minute delays caused by overhead, SIMPLE calculates the time spent on processing faults and subtracts it from the calculated fault-injection time. Overhead incurred by SIMPLE represents artificially-induced *dead-time* that should not be considered when determining fault-injection time.

8. Affect of Compiler-Induced Optimizations

Since compiler-induced optimizations can change the resulting byte-code and thus affecting source-level debugging, SIMPLE is not guaranteed to work with Java classes compiled with the optimization option (i.e., the `-O` switch) of the `javac` compiler tool.³⁰

9. Software Fault Evaluation is Coarse-Grained

SIMPLE does not currently provide a built-in monitoring capability to track, record, and analyze SUT execution during fault-injection testing. Nor does it consist of a

²⁸ Imagine further compounding the problem by inserting another fault within the loop.

²⁹ These events are typical in conventional debuggers.

³⁰ The Sun Java 2 SDK does not currently provide an implementation for the `-O` option.

data-collection capability. Such capabilities would be especially useful during post-analysis tasks, such as comparing data sets generated from different fault-injection test sessions. However, a monitor mechanism would increase the resource overhead already consumed by SIMPLE.

Unfortunately, previous attempts to integrate a run-time statement coverage mechanism into SIMPLE proved too prohibitive due to the overhead it incurred. Specifically, we attempted to utilize step events to record which source statements were executed during run-time. Depending on how it is configured, Step events can be fired for each program statement that is executed in the JVM [39]. Hence, a simple coverage scheme would be to record all the statements that correspond to each Step event. Unfortunately, imagine the cumulative impact that this coverage mechanism would yield in the code example in Figure 6, where a Step event is generated for each Java statement executed in the debuggee JVM.

Therefore, the lack of a coverage mechanism reduces software fault evaluations to solely be based on observations reported by the tester during fault-injection testing. Some relevant observations include the following: exceptions thrown, incorrect software behaviors, faults handled correctly, faults handled incorrectly, and visible signs of performance degradation. One caveat is that the tester must have some working familiarity with the software in order to make appropriate assessments concerning the software. Unfortunately, this brute-force and time-consuming approach can be very tedious and coarse-grained. Moreover, it is prone to tester-error [27].

10. Interaction with Other Software Tools

Since SIMPLE exclusively launches the SUT in a separate JVM debug process, this may limit concurrent collaboration with other software tools that also launch the SUT in their own separate processes. Hence, it may not be possible for SIMPLE and other software tools to simultaneously manipulate the same SUT process. Hence, testing using the various tools in conjunction with SIMPLE should occur separately in different test sessions.

However, this is not to say that it is impossible for collaboration between such tools to occur. For instance, it is possible to use the output of SIMPLE to serve as input to

other tools, and vice versa. In this manner, no interference between test processes would exist and information from one test tool could be passed on to the other. Section D of Chapter VII documents some of our efforts using a coverage tool along with SIMPLE.

G. FUTURE ENHANCEMENTS

This section describes some future work proposed for SIMPLE.

1. Using Perturbation Functions to Improve Variable-Mutation Performance

Despite the problems discussed earlier in this chapter, byte-code pre-instrumentation works relatively well in SIMPLE. For instance, embedded SUT faults can be natively executed on the host JVM with minimal intervention from SIMPLE. Another reason is that pre-instrumentation would eliminate the need for a remote test-harness such as SIMPLE, thereby increasing SUT performance and responsiveness. Recall that debugger communication between JVMs can be quite expensive. For these reasons, a full pre-instrumentation solution to fault-injection would be worthwhile to pursue for enhancing the testing process.

The variable-mutation faults are not currently pre-instrumented in the SUT. Recall that we chose to utilize Sun's JPDA technology to help implement them. However, an alternative approach would be to hard-code these faults using embedded perturbation functions. Perturbation functions are the "source code instrumentation utilities" used to mutate the data-states of selected program variables [3]. Technically speaking, these functions can be used to change the values of certain SUT variables during run-time.

```
1 public int perturb(int a, boolean activate)
2 {
3     if (activate)
4     {
5         return rand(a);
6     }
7     return a;
8 }
```

Figure 7. Perturbation Function

Figure 7 shows an example of a simplistic perturbation function that can mutate an integer variable. Basically, the function accepts the variable to be mutated as an argument to the method and returns a new value based on logic within the function.

Perturbation functions can be readily integrated into the SUT, provided that the source code is available for annotation and recompilation. (Again, source code is needed in order to replace variable-access code to corresponding perturbation method invocations. After the appropriate annotations have been made, the source code is then recompiled in to a new application build.) In some respects, perturbation functions would have enhanced SIMPLE by eliminating reliance on a debugger-compliant JVM. However, in the absence of a source code annotation tool, integrating these functions would require the tester to manually implement the perturbation functions and modify the source code accordingly. While not necessarily a bad thing per se, this approach would not be very automatic or transparent to the tester.

In a manner similar to the pre-instrumentation technique employed by SIMPLE, future work would entail investigating ways to automatically define, customize, and insert specialized perturbation functions into the SUT. For example, one possible approach would be to construct a source-level parser (e.g., a JavaCC³¹ parser) that replaces variable names in the source code with an appropriate perturbation function invocation.

2. Extending the Fault Range of SIMPLE

We described in Section C the range of faults that SIMPLE can inject into an application. These currently include variable-mutation, memory-exhaustion, processor-exhaustion, forced-delay, and exception faults. Future work would further extend this repertoire to include communication-related faults, which are useful in testing and evaluating message-based distributed software. (The Java Messaging Service by Sun, for example, is a popular message-based protocol commonly used in distributed Java applications [44].) Some examples of communication faults include the following: loss of incoming messages, loss of outgoing messages, corrupted message content, delayed message delivery, and duplicated message delivery [45]. Further work also includes the

³¹ JavaCC is a parser generator for use with Java applications [46].

implementation of additional infrastructure needed to support communication fault injections.

3. Mutating Collections and Arrays

SIMPLE does not support the mutation of dynamic collection data structures and fixed allocated arrays. Fortunately, the JPDA API does provide underlying support for accessing and modifying array structures. Hence, this capability will be provided in future versions of SIMPLE.

4. Data Collection in SIMPLE

Recall that SIMPLE does not proffer monitoring and data-collection services to support fault-injection post-analysis. This is currently a severe limitation. One future task would be to integrate such a module so that a run-time event-log is transcribed for each SUT test run. The event-log, for example, would record events such as faults injected, exceptions thrown, code coverage, and memory usage. Moreover, recorded event timestamps would especially be useful in determining the more difficult fault-injection metrics as fault latency and fault propagation.

5. Developing a GUI for SIMPLE

SIMPLE can be further improved by adding in a graphical-user interface (GUI). Many existing SWFI tools (e.g., GOOFI [21] and DOCTOR [22]) provide a user-friendly interface to facilitate the user's understanding of the system's functionality. Currently, SIMPLE provides users with only primitive, textual output from the SUT. Thus, we had to rely on "spot-checks" of program code to verify results. In addition, we were unable to debug or trace back through the interactions or behaviors that led to a particular event (or fault). Using a COTS coverage tool somewhat helped on verifying the location of the injected fault. Still, we were unable to trace back calls after fault injection to possibly discover other parts of vulnerable code.

For the next version of SIMPLE, we plan to implement a GUI to provide the following features:

- Allow for users to interactively define and inject faults during run-time, thus eliminating the need for batch-files or start-up scripts.

- Incorporate a debugger that enables users to step through code and inject faults at pre-determined break points.
- Have an option to display coverage showing both covered and uncovered code after a fault injection has been made.
- Enable users to interactively select fault injection models.
- A log screen that displays interactions between SIMPLE and the SUT.
- A separate screen that shows SUT output.
- A results screen that displays statistical data and relevant metrics.
- A dialog window to allow testers configure fault attributes (i.e., location, time) during run-time.

6. Further Investigation of Java Technologies

ProbeMeister by Object Services & Consulting, Inc. claims to provide the capability for dynamically inserting and removing byte-code into an application during run-time [47]. This would certainly improve upon SIMPLE's current fault pre-instrumentation feature that requires all class modifications to be persisted (i.e., saved to the disk) before execution. In addition, this technology could provide the basis for implementing a built-in code-coverage or data-collection capability for SIMPLE. What's more, *ProbeMeister* also claims not to require source code. This is also highly desirable since it would relax our source code requirement.

7. Further Investigation of Open-Source Fault-Injection Tools

The *SourceForge* website³² contains various open-source Java fault-injection tools that are publicly available for download, such as *FIDe* (Fault Injection via Debug) [67], the *Linux Fault Injection Test Harness* project [68], and *JPWrite* (Network Fault Injection System for Java) [69]. Follow-on work for SIMPLE would entail close examination of these fault-injection engines.

³² SourceForge website available at <http://sourceforge.net/>, 2002.

H. PLAN TO THROW ONE AWAY³³

In its current form, SIMPLE is not yet an industrial-strength software testing tool. Rather, it represents an implementation of accumulated ideas newly introduced to us during our software fault-injection research. As to the prototypical nature of SIMPLE, we defer to Frederick Brooks as he states in *The Mythical Man-Month* [65]:

[The first system] may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved ... Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time.

Therefore, SIMPLE is at best a skeletal testing tool that needs to be improved upon for robustness, usability, and for other requirements that are presently unknown to us. In this regard, we heed Brook's advice and anticipate using our lessons-learned and experiences to construct a more robust version of SIMPLE. Our motivation to continue development is given by example in the case studies chapter.

³³ This section is titled after a chapter of *The Mythical Man Month* by Fredrick P. Brooks, Jr. The chapter reference refers to the prototypical nature of SIMPLE.

VII. CASE STUDIES

A. INTRODUCTION

Upon the completion of SIMPLE, several case studies were immediately conducted to evaluate the usability and effectiveness of SIMPLE.³⁴

Section B of this chapter describes how SIMPLE can verify test cases generated for unit-level testing. In particular, SIMPLE assessed unit tests developed for a *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) LAN discrete-event simulation program. Section C documents how SIMPLE uncovered software frailties in our prototype of an airline reservation system (ARS). In addition, the notion of fault acceleration is demonstrated in this section. Section D documents how SIMPLE effectively increased test coverage during software testing.

B. CASE STUDY I: USING SIMPLE TO VERIFY TEST CASES

Mutation testing provides a systematic means to evaluate *test-case adequacy* [10]. As demonstrated later, mutation testing is useful for verifying unit- and integration-level testing strategies. Specifically, this type of testing involves creating *mutant programs* to corrupt program state. Thus, mutant-induced errors can evaluate test case fault sensitivity. As Voas mentions in [3], “Mutation testing attempts to see how good test cases are at detecting injected anomalies.” For instance, if the test case detects anomalous behaviors caused by the mutant program, then this test case is very effective. On the other hand, if the test case misses the error, then it is deemed ineffective for testing. Fortunately, those test cases found to be inadequate can be corrected in time for testing. Thus, this verification process improves the overall reliability of our test cases. Remember, bugs that escape the testing phase due to a faulty test plan will in general be more expensive to fix in later developmental stages [48].

SIMPLE can be an effective mechanism for mutation testing and test case assessment. Test cases can be verified using fault-injection to ensure their effectiveness and adequacy. As an illustration, SIMPLE was used to verify a number of test cases previously generated for the CSMA/CD simulation program.

³⁴ Simultaneously, these case studies also have served to test, troubleshoot, and debug SIMPLE.

1. CSMA/CD Software Description

In their network utilization study, Sadiku and Ilyas implemented software to simulate a local area network that utilizes *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) access protocol [49]. The CSMA/CD software simulates the processing of data packets communicated between client workstations. It attempts to model a realistic network transmission medium by allowing “collisions” to occur during message transportation. In effect, this simulates random interruptions and/or disturbances in network traffic flow. During the simulation, network-centric measurements are generated, such as packet delivery delay, data throughput, and collision frequency. The resulting recorded metrics are displayed to the user once the simulation run is complete.

Our case study used an object-oriented version of the CSMA/CD simulation software. The software was revised from its original form to improve its testability.³⁵ See Figure C-1 in Appendix C for a class diagram of the object-oriented version of the software.

2. JUnit Framework

JUnit is an open-source, regression-testing³⁶ framework written by Erich Gamma and Kent Beck [50]. The framework provides an infrastructure for rapidly building test cases and test suites for an application. More importantly, it allows for the automatic execution of test cases during unit-, integration-, and regression testing. This lightweight mechanism supports an incremental process of rapid software design and development, as proposed in the Extreme Programming methodology [51]. See Figure C-2 for a high-level class diagram of the JUnit Framework.

The JUnit framework supported our unit- and integration-testing for the CSMA/CD simulation software. A JUnit test suite yields a composite structure that facilitates the development of large, complicated test suites. For instance, a test suite can itself consist of many test cases, or it can contain other test suites. In other words, a test

³⁵ The CSMA/CD software was an assigned software-testing project for a Naval Postgraduate School (NPS) Software Engineering course, *SW4540: Software Testing*.

³⁶ *Regression testing* refers to the selective retesting of software components. This is primarily to ensure that any recent software changes haven’t adversely affected those components [27].

suite can be composed of both test cases and test suites. In addition, the framework is geared toward executing these test suites in an automatic and repeatable manner.

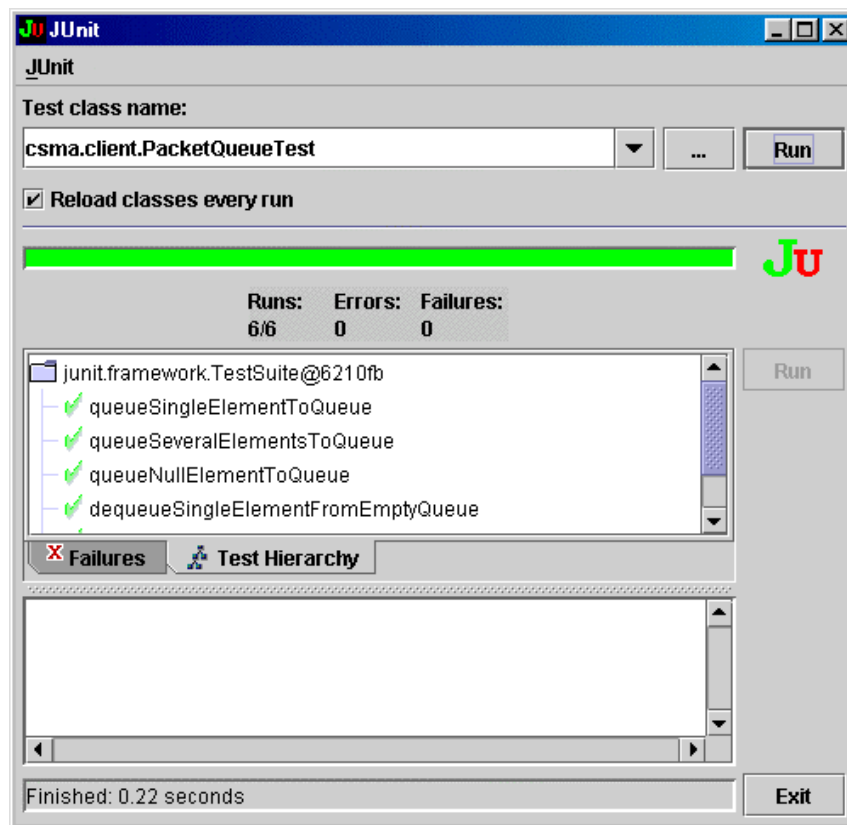


Figure 8. JUnit GUI

The JUnit GUI shown in Figure 8 allows testers to select test suites, execute test cases, and view test results. The progress status bar shown in the GUI graphically displays the percentage of tests completed during testing. If all of the tests within the test suite execute without error, the progress status bar turns green. Otherwise, the progress status bar turns (and remains) red to indicate that a test error has occurred.

3. CSMA/CD Test Suites

Five test suites were generated to test the CSMA/CD simulation software at the method- and class-level. The current set of test suites include the following: *NetworkSimulationMainTest*, *PacketQueueTest*, *StationTest*, *NetworkEventManagerTest*, and *NetworkTest*. Each test suite consists of test cases that exercise various classes in the CSMA/CD simulation software. For instance, the *NetworkEventManagerTest* test suite

utilizes test cases that specifically evaluate method and class behavior of the *NetworkEventManager* class. See Figure C-3 for a class diagram of the test suites.

More general test suite descriptions are given in the following table.

Test Suite	Description
<i>NetworkSimulationMainTest</i>	This test suite provides test cases for exercising the network configuration property file feature of the CSMA/CD LAN Simulation Software. This feature allows the user to input network configuration parameters to be applied to the network simulation without recompilation. In short, different test property files will be read by each test case to determine the robustness of this parameter input mechanism. Hence, to help assess how the CSMA/CD software responds to erroneous input, the test property files used in this test suite will be corrupted or contain corrupted configuration values.
<i>PacketQueueTest</i>	This test suite provides test cases as generated from the Quasi-Modal Test Design Pattern [27], which is appropriate since the <i>PacketQueue</i> class implements a queue data type. ³⁷
<i>StationTest</i>	This test suite provides test cases that test the station's usage of the <i>PacketQueueTest</i> . Some test cases from the <i>PacketQueueTest</i> Test Suite will be duplicated here. Also, it is the <i>Station</i> class that restricts and enforces the capacity for the <i>PacketQueue</i> class.
<i>NetworkEventManagerTest</i>	This test suite provides test cases for testing the <i>NetworkEventManager</i> class. Most of the methods are simple set and get methods that are trivial to test. Other methods such as <i>getNextPendingEvent</i> and <i>getAllSimilarEventsWithSameTime</i> are not so trivial and require some preparation to setup. Due to the importance of the latter methods, the tests will be executed a number of times using randomly generated test data.
<i>NetworkTest</i>	This test suite provides test cases that process various network events, such as <i>Arrival</i> , <i>Departure</i> , and <i>Collision</i> . By using a <i>NetworkEventManager</i> instance, each test case will designate an event to be processed next. The test case passes if it determines that the <i>Network</i> instance (through a single simulation run) has processed the designated event.

Table 1. Test Suite Descriptions

³⁷ The quasi-modal class test pattern is used to test classes whose constraints on message sequence depends on a class' particular state [27].

4. CSMA/CD Test Cases

A subset of test cases was selected from each of the test suites to avoid documenting redundant cases. The following tables provide brief descriptions of all the test cases used in this case study.

#	Test Case	Description
01	<i>withinRangeNetworkParameters</i>	Parameters with valid values are read from a configuration file into the program via the <i>NetworkSimulationMain</i> class. It will be verified that the program has accepted these parameters.
02	<i>outOfRangeNetworkParameters</i>	Parameters with invalid values are read from a configuration file into the program via the <i>NetworkSimulationMain</i> class. It will be verified that the program has not accepted these parameters and that default values are used instead.
03	<i>missingNetworkParameters</i>	Parameters with invalid values are read from a configuration file into the program via the <i>NetworkSimulationMain</i> class. It will be verified that the program has not accepted these parameters and that default values are used instead.

Table 2. "NetworkSimulationMainTest" Test Cases

#	Test Case	Description
01	<i>queueSingleElementToQueue</i>	A packet is queued to the <i>PacketQueue</i> instance. It will be verified that a packet has been stored.
02	<i>queueSeveralElementsToQueue</i>	Several packets are queued to the <i>PacketQueue</i> instance. It will be verified that the packets have been stored.
05	<i>queueSingleElementToNonEmptyQueue</i>	A packet is dequeued from a pre-loaded <i>PacketQueue</i> instance. It will be verified that a dequeued packet is the same packet that was queued beforehand.
06	<i>queueSeveralElementsToNonEmptyQueue</i>	Several packets are dequeued from a pre-loaded <i>PacketQueue</i> instance. It will be verified that the dequeued packets were the exact packets that were queued beforehand.

Table 3. "PacketQueueTest" Test Cases

#	Test Case	Description
04	<i>queueSingleElementToFullCapacityQueue</i>	A packet is queued to a <i>Station</i> instance with a full queue. It will be verified that an overflow exception has been thrown.
05	<i>queueSeveralElementsToNonEmptyQueue</i>	A station's buffer is filled to its maximum number of elements. It will be verified that the station's buffer has reached the maximum number of elements.
06	<i>queueSeveralElementsToNearCapacityQueue</i>	Several packets are queued to a <i>Station</i> instance at near full capacity. It will be verified that an overflow exception has been thrown.
10	<i>dequeueSingleElementFromNonEmptyQueue</i>	A packet is queued and immediately dequeued from a <i>Station</i> instance. It will be verified that the dequeued packet is the same packet that was queued beforehand.
12	<i>testStationSetAndGetIdMethods</i>	The station ID will be set via the <i>setID</i> method. It will be verified that the ID returned from the <i>getID</i> method is the same ID that was set beforehand.

Table 4. "StationTest" Test Cases

#	Test Case	Description
01	<i>testSetAndGetEventTimeMethods</i>	It will be verified that various set and get methods of the <i>NetworkEventManager</i> class are operating correctly. For instance, the value returned from the get method should match the same value that was set beforehand with the corresponding set method.
02	<i>testSetAndClearEventTimeMethods</i>	It will be verified that the clear methods of the <i>NetworkEventManager</i> class are operating correctly. For instance, a <i>clearArrivalEventTime</i> invocation should reset the <i>eventTime</i> of the <i>Event</i> instance corresponding to an <i>Arrival</i> event.
03	<i>testGetEventWithSmallestTimeMethod</i>	It will be verified that the <i>getNextPendingEvent</i> method of the <i>NetworkEventManager</i> class is operating correctly. More specifically, the <i>getNextPendingEvent</i> method should return the <i>Event</i> instance with the smallest event time.

Table 5. "NetworkEventManagerTest" Test Cases

#	Test Case	Description
01	<i>verifyProcessingOfArrivalEvent</i>	The <i>NetworkEventManager</i> is configured so that an <i>Arrival</i> event will have the lowest time. It will be verified that the <i>Network</i> instance has processed the correct <i>Arrival</i> event.
02	<i>verifyProcessingOfTransAttEvent</i>	The <i>NetworkEventManager</i> is configured so that a <i>TransmissionAttempt</i> event will have the lowest time. It will be verified that the <i>Network</i> instance has processed the correct <i>TransmissionAttempt</i> event.
03	<i>verifyProcessingOfCollChkEvent</i>	The <i>NetworkEventManager</i> is configured so that a <i>CollisionCheck</i> event will have the lowest time. It will be verified that the <i>Network</i> instance has processed the correct <i>CollisionCheck</i> event.
05	<i>testRhoGreaterThanOrEqualToOneException</i>	It will be verified that a defined arrival rate value of 200.0 will calculate a <i>rho</i> value over 1.0, thus invoking an exception.

Table 6. "NetworkTest" Test Cases

5. Employing Fault-Injection

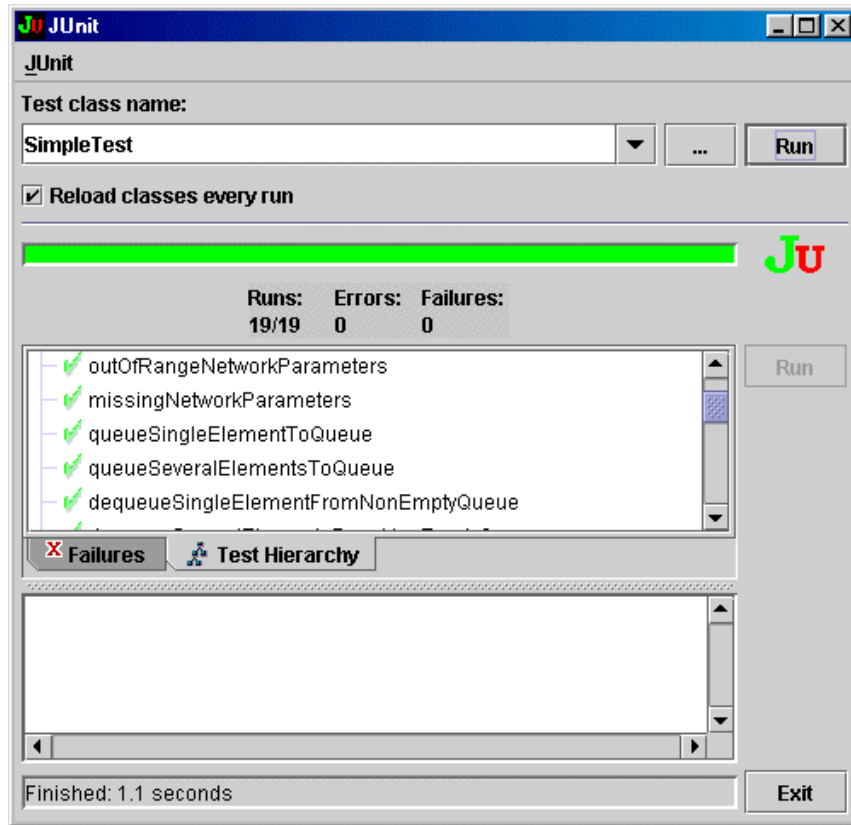


Figure 9. SimpleTest – Tests All Passed

For convenience, all of the test cases described in the previous tables were placed in a single specialized test suite called *SimpleTest*. Figure 9 pictures a JUnit GUI screenshot showing the completed execution of the *SimpleTest* test suite. Notice in the screenshot that all of the test cases have passed. (In Figure 9, recall that the progress bar is colored green to denote that all tests has passed.) This should be the case since the software has already been subjected to as series of testing iterations in a previous study.³⁸

For the purposes of this case study, program faults were injected using SIMPLE to verify the “correctness” of the test cases. The fault will depend on the test case being evaluated. (For example, programmer errors relevant to a *Stack* data structure would relate to erroneous handling of underflow and overflow scenarios.) Of course, the intention of the fault is to force a program error or failure. As noted earlier, if a test case

³⁸ As mentioned previously, the CSMA/CD software was an assigned software-testing project for a Naval Postgraduate School (NPS) Software Engineering course, *SW4540: Software Testing*.

fails to report the program error, then the test case itself is faulty and not suited for unit testing. If this happens, then SIMPLE has exposed a problem with the prior test process. Furthermore, this indicates a lack of integrity of any testing conducted previously on the SUT. However, in the real world, if caught early enough in the pre-testing phase, then the mistake might be easily remedied without affecting the project budget. At this stage, bugs would not be deeply ingrained in the early development effort, hence they are easy to fix. Bugs found at later development stages tend to be propagated from earlier stages and become deeply entrenched into the software build; these bugs are much more difficult to fix.

The following tables summarize the type of faults that were injected into each of the documented test cases. In each instance, typical software bugs that could conceivably occur during development were simulated. Appendix D-1 lists the contents of the fault configuration file used in this case study.

#	Test Case	Fault
01	<i>withinRangeNetworkParameters</i>	Corrupt an arbitrary parameter so that it is different than what is expected. This error implies a problem within the <i>NetworkSimulationMain</i> class.
02	<i>outOfRangeNetworkParameters</i>	Corrupt an arbitrary parameter so that it is different than what is expected. This error implies a problem within the <i>NetworkSimulationMain</i> class.
03	<i>missingNetworkParameters</i>	Corrupt an arbitrary parameter so that it is different than what is expected. This error implies a problem within the <i>NetworkSimulationMain</i> class.

Table 7. "NetworkSimulationMainTest" Faults

#	Test Case	Fault
01	<i>queueSingleElementToQueue</i>	Corrupt the incoming packet so that packet will not be queued. This error implies that a problem exists in either the <i>Packet</i> or the <i>PacketQueue</i> class.
02	<i>queueSeveralElementsToQueue</i>	Queue fewer packets than what is originally intended. This error implies that a problem exists in the <i>PacketQueue</i> class.
05	<i>queueSingleElementToNonEmptyQueue</i>	Change attributes of dequeued packet to values not expected by the test. This error implies that a problem exists in either the <i>Packet</i> or <i>PacketQueue</i> class.
06	<i>queueSeveralElementsToNonEmptyQueue</i>	Change attributes of dequeued packet to values not expected by the test. This error implies that a problem exists in either the <i>Packet</i> or <i>PacketQueue</i> class.

Table 8. "PacketQueueTest" Faults

#	Test Case	Fault
04	<i>queueSingleElementToFullCapacityQueue</i>	Change the maximum packet setting for the station's packet queue so that an overflow does not occur. This error implies that a problem exists in the <i>PacketQueue</i> class.
05	<i>queueSeveralElementsToNonEmptyQueue</i>	Change the maximum packet setting for the station's packet queue so that queue is not filled to the maximum. This error implies that a problem exists in the <i>PacketQueue</i> class.
06	<i>queueSeveralElementsToNearCapacityQueue</i>	Change the maximum packet setting for the station's packet queue so that an overflow does not occur. This error implies that a problem exists in the <i>PacketQueue</i> class.
10	<i>dequeueSingleElementFromNonEmptyQueue</i>	Corrupt the attributes of dequeued packet. This error implies that a problem exists in the <i>Packet</i> class.
12	<i>testStationSetAndGetIdMethods</i>	Corrupt the attributes of the <i>Station</i> instance. This error implies that a problem exists in the <i>Station</i> class.

Table 9. "StationTest" Faults

#	Test Case	Fault
01	<i>testSetAndGetEventTimeMethods</i>	Corrupt the value retrieved from a get method so that it is different than what is expected. This error implies a problem within the <i>NetworkEventManager</i> class.
02	<i>testSetAndClearEventTimeMethods</i>	Corrupt the value retrieved from a get method so that it is different than what is expected. This error implies a problem within the <i>NetworkEventManager</i> class.
03	<i>testGetEventWithSmallestTimeMethod</i>	Corrupt the value retrieved from a get method so that it is different than what is expected. This error implies a problem within the <i>NetworkEventManager</i> class.

Table 10. "NetworkEventManagerTest" Faults

#	Test Case	Fault
01	<i>verifyProcessingOfArrivalEvent</i>	Modify program to always indicate that an <i>Arrival</i> event was never processed. This error implies a problem within the <i>Network</i> class.
02	<i>verifyProcessingOfTransAttEvent</i>	Modify program state to always indicate that a <i>TransmissionAttempt</i> event was never processed. This error implies a problem within the <i>Network</i> class.
03	<i>verifyProcessingOfCollChkEvent</i>	Modify program state to always indicate that a <i>CollisionCheck</i> event was never processed. This error implies a problem within the <i>Network</i> class.
05	<i>testRhoGreaterThanOrEqualToOneException</i>	Modify the arrival rate value so that it calculates a <i>rho</i> value less than one. This error implies that a problem exists within the <i>Network</i> class.

Table 11. "NetworkTest" Faults

6. Results

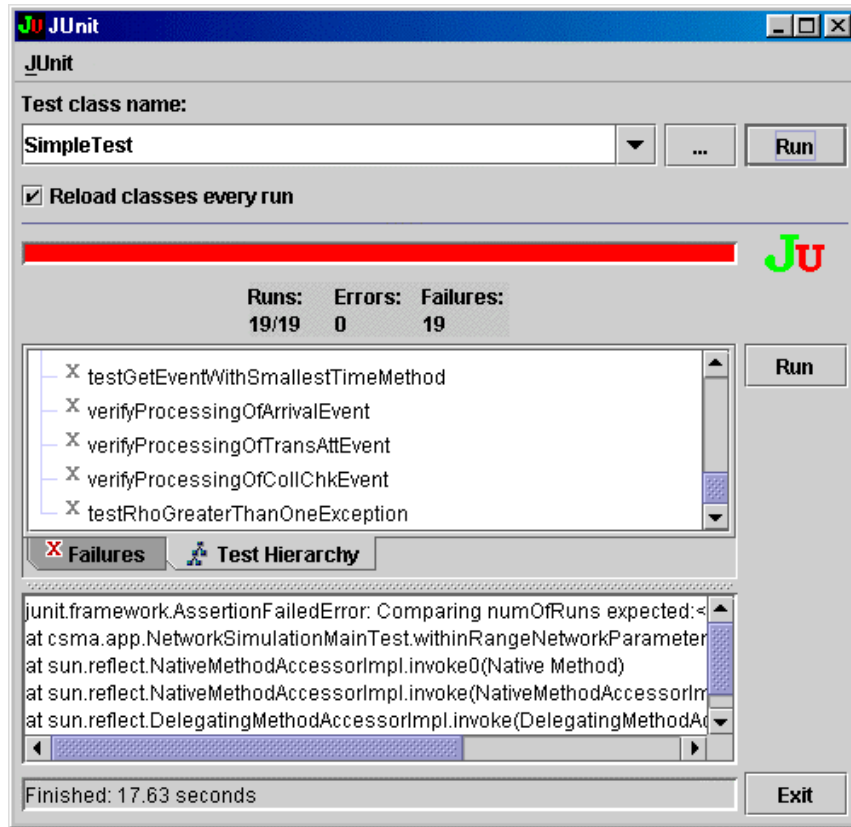


Figure 10. SimpleTest – Tests All Failed

Figure 10 shows a completed test run of the SimpleTest test suite. When triggered, SIMPLE injects the aforementioned faults during each test case execution within the JUnit GUI. As indicated in the figure, all of the test cases have failed. (In Figure 10, the progress bar is colored red to denote that tests have failed.) Thus, the test cases were sensitive enough to catch the injected errors.

7. Discussion

As demonstrated, SIMPLE can be used as a vehicle to judge test-case adequacy. Unfortunately, “test-case” testing suffers from some of the same problems currently plaguing software testing. That is, how extensive do you test your test cases? What real-world faults do you choose to inject? How many of them do you inject? Since mutation testing generates an enormous number of mutant programs, which mutant programs do you actually consider during testing? How do you determine criteria for completion? More importantly, what criteria do you use for determining test-adequacy?

For simple unit-level test cases, these questions may not be too difficult to address. For example, in our CSMA/CD case study, fault injection was able to verify our unit test cases due the simple nature of our test cases. That is, the criteria for test case verification were made trivial due to test case simplicity. On the other hand, the aforementioned questions become less obvious for testing strategies that incorporate multiple software component interactions. For example, verifying integration- and system-level test cases would not be trivial. Depending on the implementation, higher-level test cases can be much more involved, complex, and sophisticated than unit-level test cases. In addition, the individual software components that were rigorously tested via unit-level test cases will not guarantee that correctness will be shown in integration- or system-level testing. Recall that the anti-decomposition axiom in [27] tells us “... a test suite that covers a class or a method does not necessarily cover the server objects of that class or method.” Thus, a different criterion for test case adequacy is required for various types of testing strategies. Verifying these test cases would probably be just as involved as testing the software itself. These issues are beyond the scope of this thesis. These issues have been addressed to some extent in [52, 53, 54].

In summary, it is not our intent to propose a complete test-case verification process in this case study. Rather, we seek to illustrate how a fault-injection engine, such as SIMPLE, could be used to conceivably develop such a process.

C. CASE STUDY II: UNCOVERING SOFTWARE ANOMALIES USING SIMPLE

SIMPLE can facilitate software development by serving as a specialized debugging tool that offers fault-injection capabilities. Its premise is straightforward and concise: Inject faults into functional components of the software, and evaluate its resulting responses (or lack thereof) for correctness. For example, faults are forced into the software in an attempt to expose other faults. Hence, in this respect, the notion of fault-acceleration is subscribed into our test process. This section discusses some of the faults that were injected into the ARS via SIMPLE for determining robustness and fault resiliency.

This case study illustrates how faults were injected into exception-handling code. The motivation for this type of testing resulted from the fact that the original ARS test

plan did not contain any procedures that exercised the ARS fault-tolerance mechanisms. Thus, SIMPLE was used to force difficult-to-reach program paths such as exception-handling code [10].

In order to determine software resiliency, we later discuss how SIMPLE simulates a time-consuming task within the ARS. Additionally, the effect this fault has on the ARS is also discussed.

1. The Airline Reservation System (ARS) Software Description

The ARS is primarily a GUI-driven, database application for managing flight, customer, reservation, and cancellation data.³⁹ The system was specifically designed for travel agents and flight managers. The relational database is an integral part of the ARS because it stores the persistent information of the ARS.

The ARS responds to data requests from the travel agent or flight manager by displaying the requested data to the screen. An ARS GUI is tailored according to the defined role of the current system user: travel agent or flight manager. (The role of the user is authenticated during ARS login.) When inputs to the ARS are invalid, the user will be notified of the error. The ARS assigns specific operational functions to each user. For example, the ARS allows the travel agent to request flight information and make a reservation. If a reserved flight has been modified, the travel agent may decide to cancel reservations on the modified flight. The flight manager is exclusively allowed to add, delete, or modify flights. The flight manager may delete an entire flight, even if there are existing reservations. The ARS will notify the travel agent when a customer's reservation has been affected. The ARS restricts some modifications the flight manager may make. For example, the flight manager may not "bump" off customers from a flight by reducing the number of seats to a value that is less than the current number of flight reservations. If the flight manager modifies the fare of a flight, previously made reservations will not be affected. The ARS will generate an E-ticket number for each reservation made.

See Figure C-4 for a class diagram of the ARS System.

³⁹ The ARS software was another assigned project for a Naval Postgraduate School (NPS) Software Engineering course, *SW3460: Software Methodologies*.

2. Testing the ARS Exception-Handling Capabilities

During software testing conducted well prior to this case study, the execution of the ARS test plan uncovered a myriad of software bugs in the operational setting. Unfortunately, the test plan focused on program correctness in an ideal operating environment, one that is free of external errors, for example. As a result, no test procedures were generated for exercising ARS exception-handling code. In retrospect, creating a specialized test scenario that would trigger an exception at the appropriate times would have been extremely difficult. As it pertains to the ARS, the exception-handling mechanisms are “hidden” and thus inaccessible for testing. Improving the testability of these regions would have necessitated an intensive programming effort. SIMPLE illustrates that this need not be the case by demonstrating an effective fault-injection approach on an essential ARS software component: the *DatabaseManager* class.

The *DatabaseManager* class is considered the heart of the ARS software. It encompasses integral relational database operations used for processing flight, customer, ticket, and reservation records. Typical database operations include querying, storing, retrieving, deleting and modifying dataset records. Due to their importance, these operations were extensively tested in the ARS test plan. Unfortunately, as already mentioned, previous testing did not consider its exception-handling capabilities. Thus, to illustrate how SIMPLE can access exception-handling code, SIMPLE forced an exception to occur within critical regions of selected *DatabaseManager* class methods.

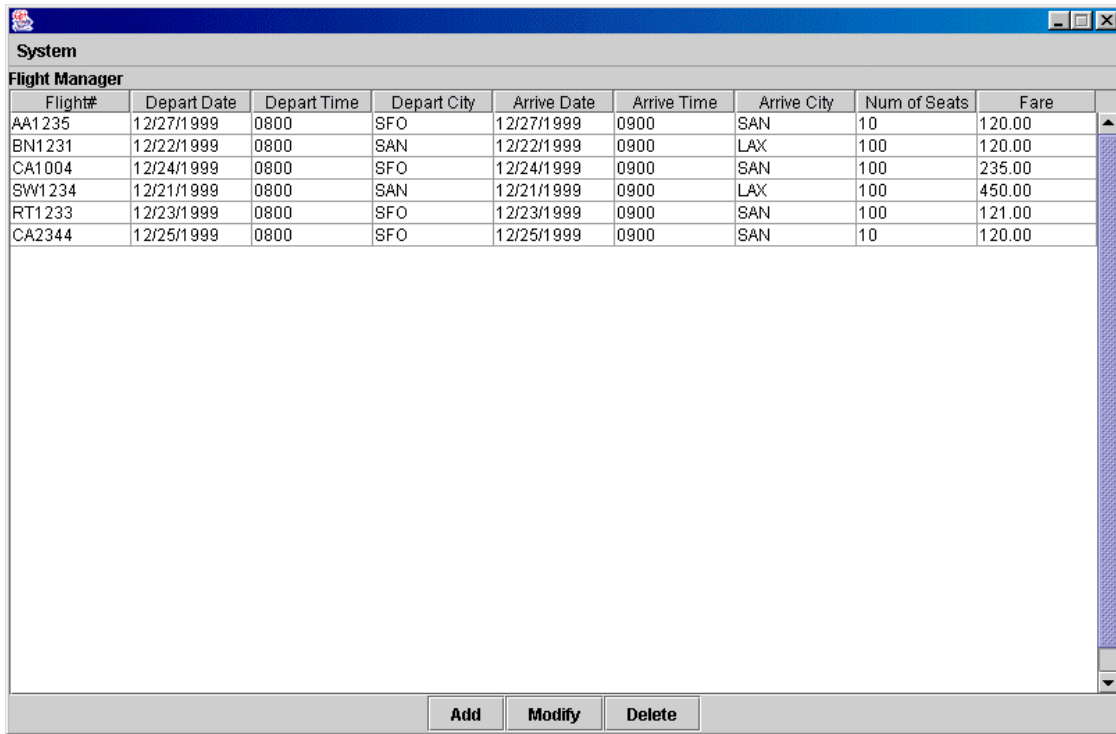
```

131
132 /**
133  * Returns the results from the database based on the specified query.
134  * @param query The query to send to the SQL Database.
135  * @returns ResultSet The Results from the query
136  */
137 private static ResultSet executeQuery(String query)
138 {
139     ResultSet result = null;
140     try
141     {
142         result = arsDBStatement.executeQuery(query);
143     }
144     catch (Exception e)
145     {
146         System.out.println("executeQuery " + e.getMessage());
147     }
148     return result;
149 }
150
151 /**
152  * Executes an update SQL query.
153  * @param query The query to send to the SQL Database.
154  * @returns boolean success/fail of execution
155  */
156 private static boolean executeUpdate(String query)
157 {
158     try
159     {
160         arsDBStatement.executeUpdate(query);
161     }
162     catch (Exception e)
163     {
164         System.out.println("executeUpdate " + e.getMessage());
165         return false;
166     }
167     return true;
168 }
169

```

Figure 11. ARS Source Code Snippet

Figure 11 shows a source listing for two very important methods of the *DatabaseManager* class, *executeQuery* and *executeUpdate*. Basically, these methods are responsible for managing and processing data in the ARS database. In *executeQuery*, database results are returned (line 148) based on system-specified queries that are executed (line 142). In *executeUpdate*, a boolean value is returned (lines 165, 167) depending on the success or failure of the executed query (line 160).



The screenshot shows a window titled "System" with a sub-header "Flight Manager". Below this is a table with 9 columns: Flight#, Depart Date, Depart Time, Depart City, Arrive Date, Arrive Time, Arrive City, Num of Seats, and Fare. The table contains 6 rows of flight data. Below the table is a large empty rectangular area. At the bottom of the window are three buttons: "Add", "Modify", and "Delete".

Flight#	Depart Date	Depart Time	Depart City	Arrive Date	Arrive Time	Arrive City	Num of Seats	Fare
AA1235	12/27/1999	0800	SFO	12/27/1999	0900	SAN	10	120.00
BN1231	12/22/1999	0800	SAN	12/22/1999	0900	LAX	100	120.00
CA1004	12/24/1999	0800	SFO	12/24/1999	0900	SAN	100	235.00
SW1234	12/21/1999	0800	SAN	12/21/1999	0900	LAX	100	450.00
RT1233	12/23/1999	0800	SFO	12/23/1999	0900	SAN	100	121.00
CA2344	12/25/1999	0800	SFO	12/25/1999	0900	SAN	10	120.00

Figure 12. Flight Manager GUI

In particular, the *executeQuery* method is extensively used in the Flight Manager GUI shown in Figure 12. Via this method, the Flight Manager GUI automatically retrieves flight information records from the relational database and displays them in a tabular form. In addition, the GUI allows for flight information to be updated using the *Add*, *Modify*, and *Delete* buttons.

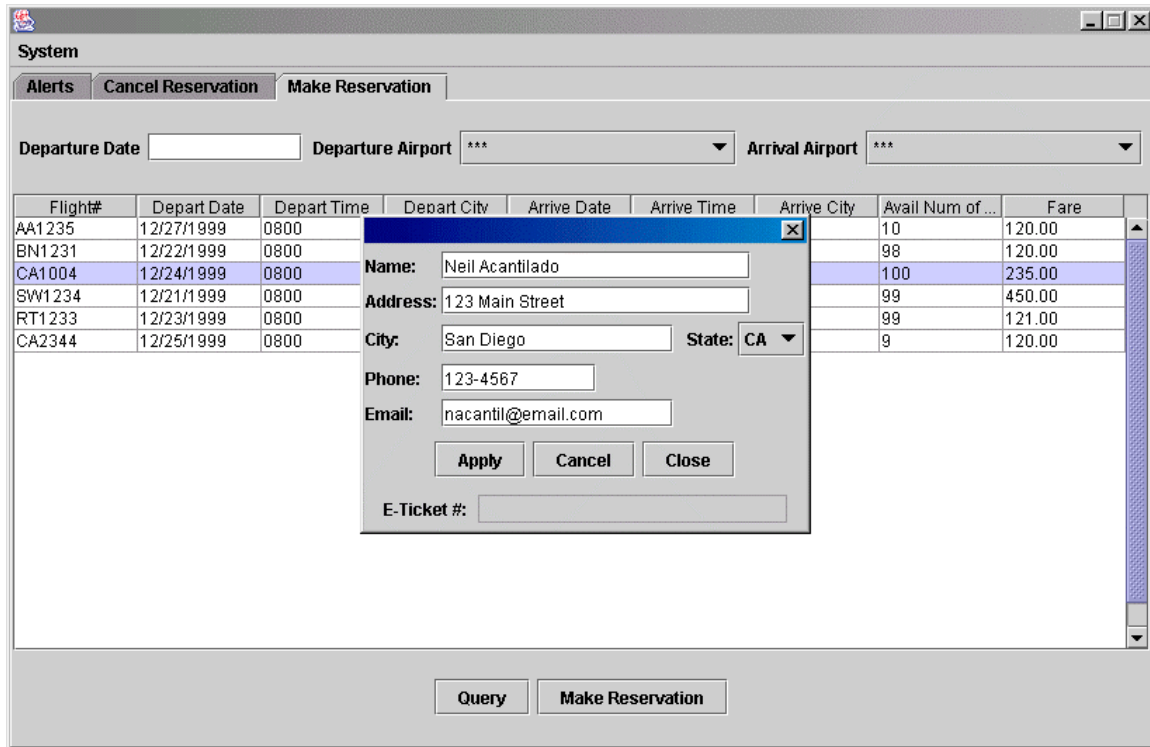


Figure 13. Travel Agent Reservation GUI with Reservation Dialog Box

The *executeUpdate* method, on the other hand, is extensively used in the Travel Agent Reservation Dialog Window GUI, shown in the foreground of Figure 13. (The GUI shown in the background of Figure 13 is the Travel Agent Reservation GUI. The Dialog Window GUI appears in the Travel Agent Reservation GUI when a flight is selected from the table and the *Make Reservation* button is pressed.) Through this method, the Dialog Window GUI creates flight reservations for the ARS system and is immediately submitted (i.e., stored into the database) when the *Apply* button is pressed.

Both *executeQuery* and *executeUpdate* methods utilize standard JDBC⁴⁰ constructs, such as *Statement* and *ResultSet*, for querying, retrieving, and affecting database information. Of primary interest, however, are the error-handling regions of these methods (lines 144-147 and lines 162-166 in Figure 11.) To evaluate their effectiveness against failure, SIMPLE was used to inject variable-corruption faults to invoke exceptions within these methods during run-time.

⁴⁰ The JDBC, which stands for *Java Database Connectivity*, provides Java API to access tabular data from virtually any data-source, such as a relational databases or spreadsheets. See [70] for more information.

In order to execute lines 144-147 in *executeQuery*, a failure has to occur on line 142 to raise an exception. Similarly, to execute lines 162-166 in *executeUpdate*, a failure on line 160 has to occur to trigger an exception. Consequently, the construct used in both lines 142 and 160 is the *arsDBStatement* variable, which is an instance of the JDBC *Statement* class. In each case, this variable was purposely “nullified” (i.e., set to null) via SIMPLE to forcefully execute the aforementioned lines of code. Appendix D-2 lists the contents of the fault configuration file used in this particular test. Again, exercising these exception calls without SIMPLE would be very difficult. In addition, it would most likely require modifications to the source code (i.e., increasing testability) to test for exception-handling. This may prove to be expensive and time-consuming.

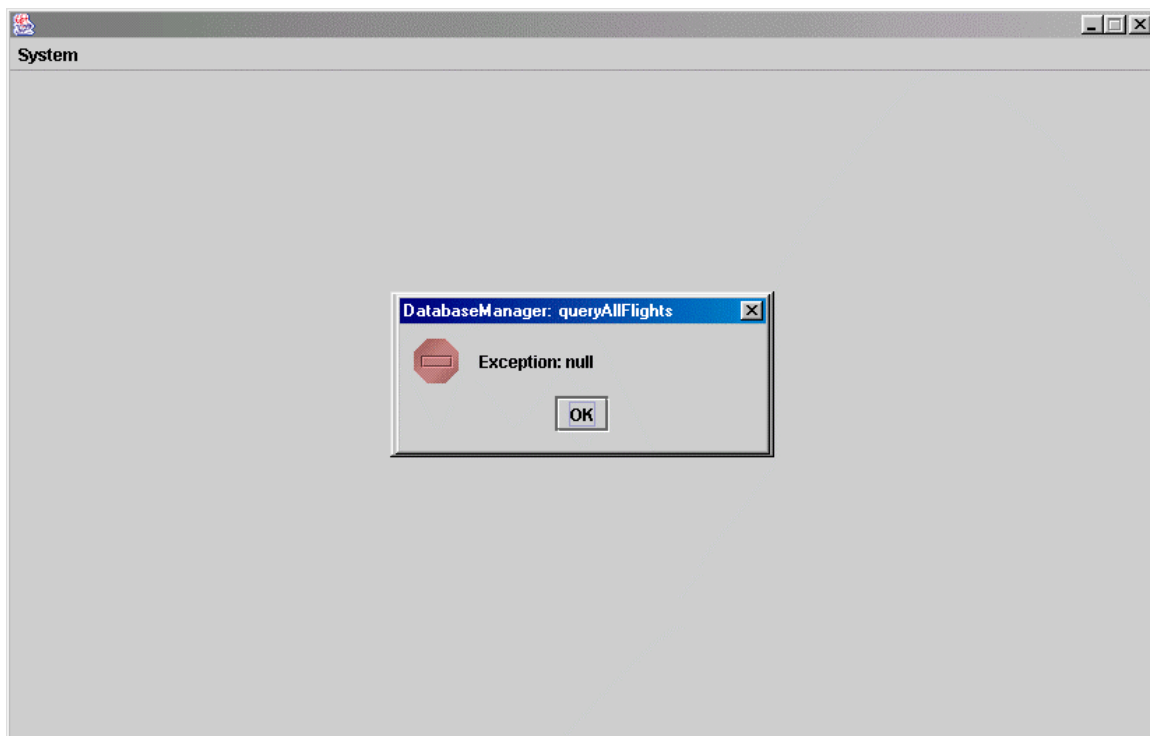


Figure 14. The Flight Manager GUI Session with Injected Fault

Figure 14 shows the Flight Manager GUI alerting the flight manager that an exception was raised. This is in response to the variable-corruption fault injected by SIMPLE in the *executeQuery* method. As a result, no flight information was retrieved. While the alert window notified the operator of the ensuing database error, it does not identify the cause of the fault. Despite this, the Flight Manager GUI responded correctly

to the encountered fault. In the future, the ARS developers may want to redesign the alert to contain more specific detail concerning the encountered error.

The screenshot shows a 'System' window with tabs for 'Alerts', 'Cancel Reservation', and 'Make Reservation'. The 'Make Reservation' tab is active. Below the tabs are input fields for 'Departure Date', 'Departure Airport', and 'Arrival Airport'. A table lists flight details, with flight CA1004 highlighted. A modal dialog box is open, displaying reservation details for Neil Acantilado, including address, city, state, phone, email, and an E-Ticket #.

Flight#	Depart Date	Depart Time	Depart City	Arrive Date	Arrive Time	Arrive City	Avail Num of ...	Fare
AA1235	12/27/1999	0800					10	120.00
BN1231	12/22/1999	0800					97	120.00
CA1004	12/24/1999	0800					99	235.00
SW1234	12/21/1999	0800					99	450.00
RT1233	12/23/1999	0800					99	121.00
CA2344	12/25/1999	0800					9	120.00

Reservation Details:

Name: Neil Acantilado
Address: 123 Main Street
City: San Diego State: CA
Phone: 123-1234
Email: nacantil@email.com
E-Ticket #: 000000010

Figure 15. A Travel Agent Reservation GUI Session with Injected Fault

On the other hand, the effect of the fault that was inserted into *executeUpdate* did not appear in the Travel Agent Reservation GUI. Despite the presence of this fault, Figure 15 shows the top-level Travel Agent Reservation GUI after a reservation has been committed via the Travel Agent Reservation Dialog Window GUI. Unfortunately, the GUI failed to indicate that a fault occurred. In fact, despite the fault, the system erroneously generated an E-Ticket number, as shown in the *E-Ticket#* field. (In other words, instead of reporting an error-message, the GUI indicates that no problems have been encountered.) Hence, in this case, it is not apparent what the total effect the fault had on the ARS system.

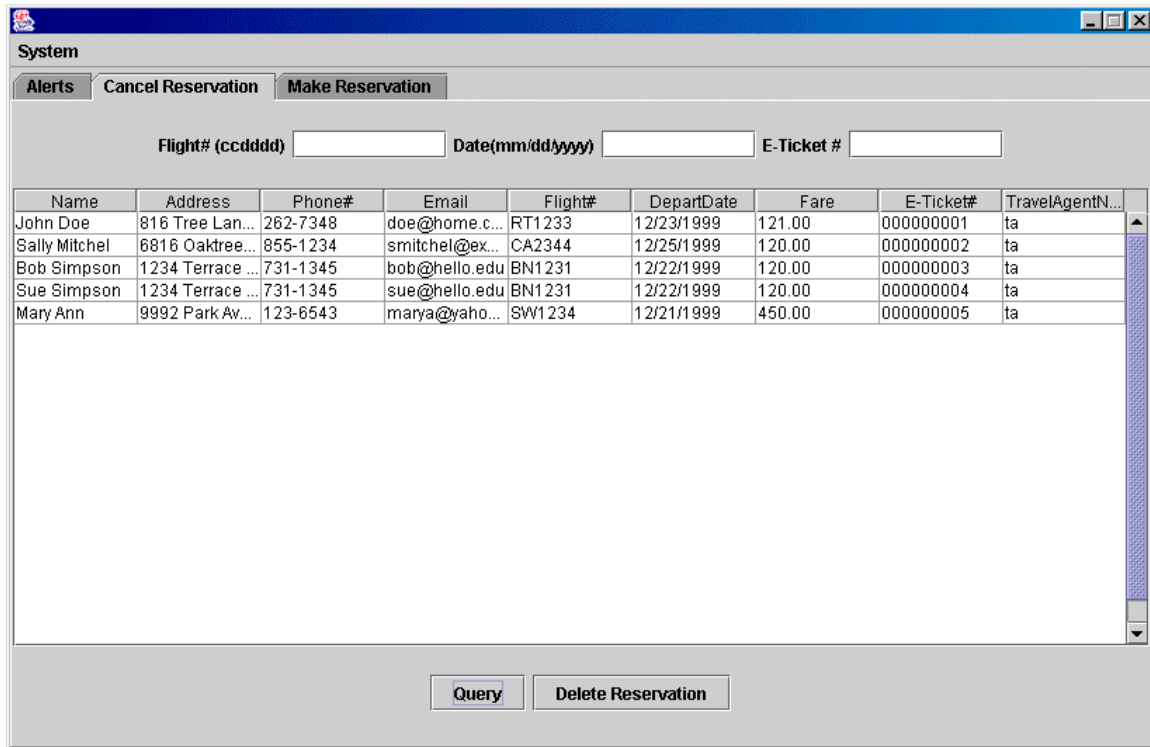


Figure 16. Reservations in the Travel Agent Reservation GUI.

Figure 16 shows a Travel Agent Reservation GUI screen displaying recorded reservations in the ARS database. Note that the reservation for “Neil Acantilado” does not exist, even though one apparently was created in the previous step (See Figure 15). Hence, SIMPLE exposed a flaw in the ARS system by applying fault injection in the *executeUpdate* method.

```

165
166     // Get ETicketNumber
167     DatabaseManager.prepareForNewBlockTransaction();
168     String eTicketNumber = DatabaseManager.generateETicketNumber();
169
170     customerInfo.setETicketNumber(eTicketNumber);
171     eTicketTextField.setText(eTicketNumber);
172
173     DatabaseManager.prepareForNewBlockTransaction();
174     DatabaseManager.insert(customerInfo);
175 }
176 }
177

```

Figure 17. Software Bug in the Travel Agent Reservation GUI Code

On further investigation, the bug occurred because the Travel Agent GUI code ignores the return value of the *DatabaseManager insert* method (shown in line 174 in Figure 17). The *insert* method internally uses the *executeUpdate* method and propagates

its boolean return value (More precisely, the *insert* method is a convenience method that “wraps” the *executeUpdate* method.) A fix would entail evaluating the boolean returned from the *DatabaseManager insert* method on line 174. If true, then the database update operation was carried out successfully. If false, then the database update operation failed and an error-message should be generated as a result. Figure 18 shows template solution (lines 175 through 180).

```
165
166     // Get ETicketNumber
167     DatabaseManager.prepareForNewBlockTransaction();
168     String eTicketNumber = DatabaseManager.generateETicketNumber();
169
170     customerInfo.setETicketNumber(eTicketNumber);
171     eTicketTextField.setText(eTicketNumber);
172
173     DatabaseManager.prepareForNewBlockTransaction();
174
175     // Check for success/failure ...
176     if (!DatabaseManager.insert(customerInfo))
177     {
178         // Do something here to alert the user that something
179         // bad has happened during a database insert ...
180     }
181 }
```

Figure 18. Template Fix

3. Assessing GUI Performance via Fault-Acceleration

SIMPLE was used to invoke the effects of a time-consuming task on the ARS system. In this manner, we implement the notion of fault-acceleration where the failure rate of a component is accelerated via fault-injection. (This, in turn, allows for thorough testing to be conducted in a controlled environment within a limited time-frame [7].) Specifically, SIMPLE simulated a database operation that retrieves a large volume of ARS data. (How SIMPLE simulates this is explained below.) This afforded us the ability to analyze GUI performance and determine if the “GUI-freezing” phenomenon [55] is a problem in the ARS.

Rather than populating the database with massive, arbitrary ARS database records, SIMPLE injected a delay fault in one of the more critical *DatabaseManager* methods. Recall that SIMPLE is capable of injecting delay faults via a byte-code pre-instrumentation feature.

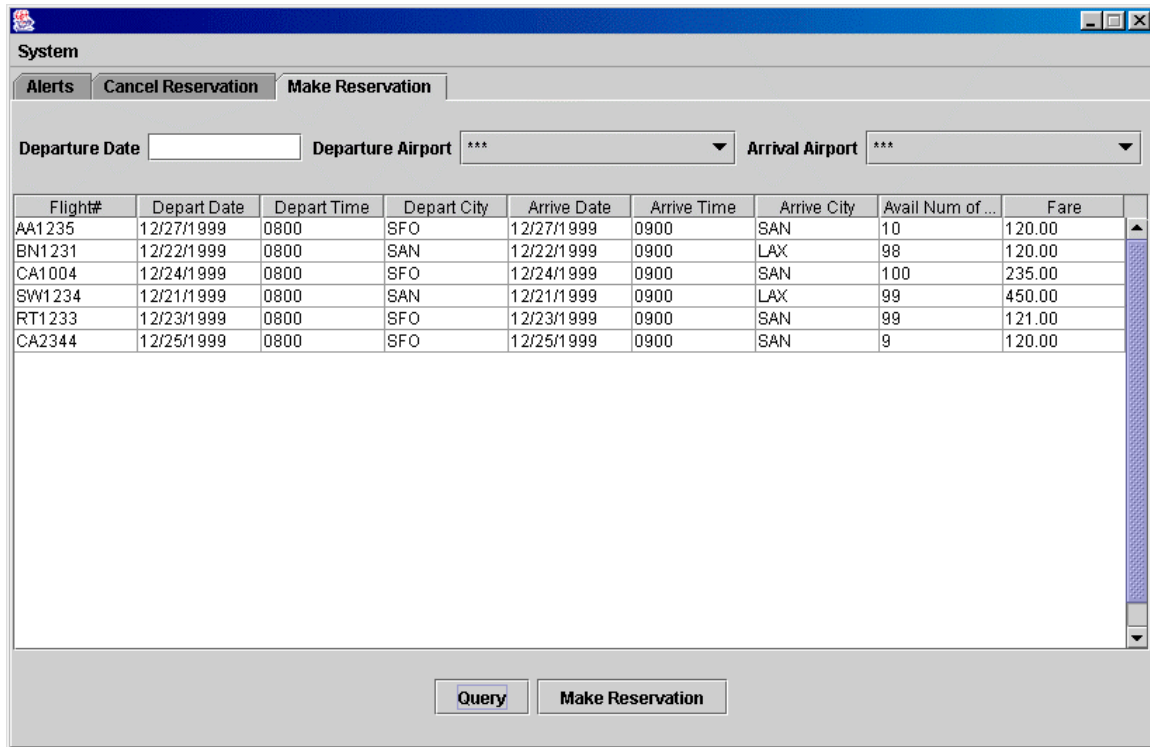


Figure 19. The Travel Agent Reservation GUI

In particular, the effect of the delay fault was examined in the Travel Agent Reservation GUI, shown in Figure 19. Appendix D-3 lists the contents of the fault configuration file used in this particular test. The Travel Agent Reservation GUI allows the travel agent to selectively filter and query flights to display. In short, system users retrieve flights via the “Query” button, based on the “Departure Date,” “Departure Airport,” and “Arrival Airport” fields. Due to its query operations, this GUI is ideal for examining GUI-related defects, such as “GUI-freezing.”

```

858         newFlightInfo.setNumOfSeats(numOfSeats);
859         newFlightInfo.setAvailNumOfSeats(availNumOfSeats);
860         newFlightInfo.setFare(airFare);
861         vector.add(newFlightInfo);
862     }
863 }
864 catch (Exception e)
865 {
866     System.out.println("query = " + e.getMessage());
867     debug(e.getMessage(), "queryFlightInfo(FlightInfo)");
868 }
869 result = null;
870 return vector;
871 }
872

```

Figure 20. Partial Listing of the Insert Method.

SIMPLE injected a delay fault into the *insert* method of the *DatabaseManager* class. Figure 20 shows a partial source listing. The fault was placed on line 870, right before the return statement. Hence, a delay occurred each time a query action is invoked within the Travel Agent Reservation GUI. This created the appropriate conditions to evaluate GUI defects.

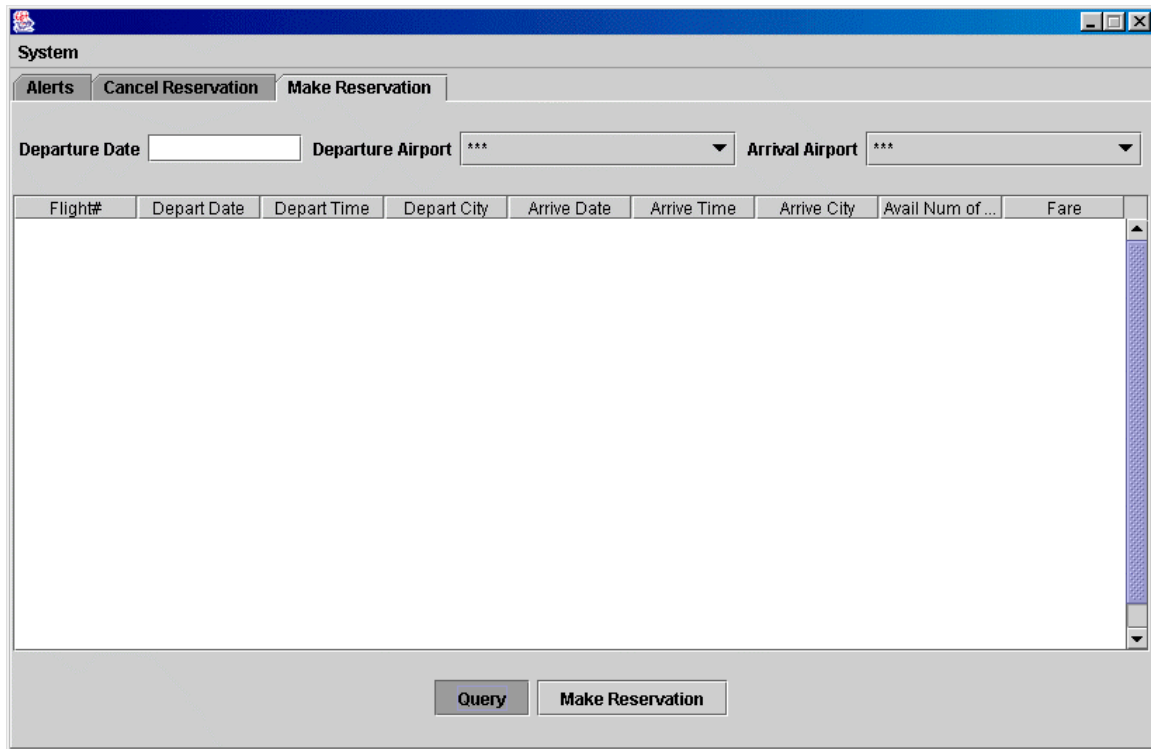


Figure 21. A "Frozen" Travel Agent Reservation GUI

Figure 21 shows the Travel Agent Reservation GUI in a “frozen” state as a result of the delay fault. Keep in mind that this is simulating a mass retrieval of information from the database. Notice the absence of a visual indicator that informs the operator that a database retrieval operation is currently underway. This is a serious GUI design flaw. (In fact, an excellent treatment of this and other similar GUI design flaws is given in [55].) Basically, the problem here is that the (simulated) database retrieval operation is not executed in a separate background thread. Thus, this adversely affects the ARS since it has to wait until the retrieval task is completed before processing other system events. In Figure 21, the GUI is in a “locked” state since it is waiting for the forced delay to complete. Hence, it remains unresponsive to user interaction. Unfortunately, the

operator may think that the ARS system has stopped functioning altogether. This could lead to further operator errors. For instance, system inactivity may cause the operator to repeatedly press GUI buttons to attempt to “unlock” the system from whatever state it is trapped in. Unfortunately, these operator actions get queued in the GUI event-queue thread to be processed later. Hence, after the delay has completed, the GUI event queue will “unravel” and process each queued GUI event. This could lead to a series of unintended database actions, such as random deletions of records. Furthermore, a frustrated operator may prematurely “kill” the application by rebooting the machine. This premature stoppage could seriously affect data integrity. To fix the problem, all potentially time-consuming tasks should be spawned into their own background threads where a dialog window or progress bar is displayed to the operator to indicate task status. Thread solutions that address time-consuming tasks are provided in [56, 57, 58].

As a result, SIMPLE exposed a potentially serious flaw in the GUI design of the ARS system through simulated fault acceleration. In other words, SIMPLE was able to expose GUI defects without having to populate the ARS database with “dummy” data.

4. Discussion

Prior ARS testing using the original test procedures successfully uncovered a multitude of software bugs. Unfortunately, due to the difficulty of reaching particular program paths, such as exception handling, were not tested. This inability to perform such paths could have had ramifications down the road had the software actually been deployed in its intended operational environment. For example, recall that the Travel Agent Reservation GUI surprisingly did not notify the operator of an underlying database problem during the creation of a flight reservation. Thus, the unsuspecting travel agent could conceivably enter multiple flight reservations before realizing that the reservations were never stored in the database. Unfortunately, this could lead to loss of work, money, and customers. Fortunately, SIMPLE uncovered this anomaly by forcing the execution of previously inaccessible exception-handling statement block; this supports our assertion that fault-injection tools should be used as complementary tools for software testing.

The preceding case study assumes developers easily have access to the application source code. For instance, inserting various faults requires access to source

code since class names and line numbers are needed. However, today's systems utilize COTS technology as cost-saving measures. As a consequence, developers no longer can inspect the underlying source code. Thus, proving software correctness becomes much more complicated. Fortunately, black-box testing techniques and strategies can help address some of these issues by testing at the system interface boundaries [27].

As an alternative approach, researchers are finding other methods to effectively test COTS products. Rather than applying black-box testing to the application, one can extend fault injection to include the operational environment of the application. For example, one can inject faults into the operating system to assess applications that are hosted on them, as was done in [1] for revealing anomalies in system behavior by COTS.

D. CASE STUDY III: INCREASING TEST COVERAGE

As with exhaustive testing, complete coverage testing is an intractable problem [27]. We will not address the problem of exit criteria for coverage testing, but rather how to facilitate this testing using fault-injection. One assertion we make is that SWFI tools can test hidden, hard-to-reach code [10]. SIMPLE illustrated this concept in the previous section. In this final case study, SIMPLE demonstrates how SWFI can increase test coverage.

1. Coverage Metrics

Code coverage analysis is an effective test strategy for “mitigating” untested code. Metrics generated from code coverage analysis can identify inaccessible code by explicitly identifying code not covered by any execution test runs. Once uncovered code is identified, specialized test stubs and drivers can then be implemented for them in order to make these areas more accessible for testing (i.e., more testable.) More importantly, coverage reports can disclose “blind spots” that the tests did not consider [27]. In effect, code coverage plays an important role in evaluating test case adequacy.

As mentioned in Section B of Chapter II, SWFI provides testers and developers with some exclusive benefits that are well suited for testing purposes. Code coverage is one such benefit. For example, if code coverage is an important exit criteria for testing completeness, then it can be shown that SIMPLE increases code coverage by executing

otherwise inaccessible regions of the software. To support this claim, an open-source coverage tool called *Gretel* was used in conjunction with SIMPLE.

2. Gretel

Gretel [60] is an open-source, Java-based, test-coverage tool developed at the University of Oregon. Unlike other coverage monitoring tools, Gretel implements *residual test coverage* monitoring, which involves instrumenting specialized probes into the application byte-code [59]. These probes record what statements were executed during run-time.

The main feature of residual test coverage monitoring is that, during re-instrumentation, instrumentation from those statements already executed are removed. This is advantageous in that it minimizes execution overhead during a rerun of the application. (For instance, statements already covered by Gretel in previous runs should not be considered again in the next run. Hence, the instrumentation for those statements are removed to avoid redundant coverage measurements.) This repeated re-instrumentation process allows for various coverage measurements compiled over various execution runs to be progressively amassed in an efficient manner. Also, information on other third-party testing tools that incorporate Gretel can be found in [61] and [62].

In short, Gretel's GUI allows testers to instrument and re-instrument selected files (i.e., Java classes) and view their corresponding coverage results after execution. Afterwards, a visualization of the source code is displayed using coverage color-codes to mark each source statement: red indicates that a statement was not executed, while green represents an executed statement.

3. Using Gretel with SIMPLE

To prove SIMPLE's effectiveness in increasing code coverage, Gretel acquired two coverages from two separate ARS execution runs.

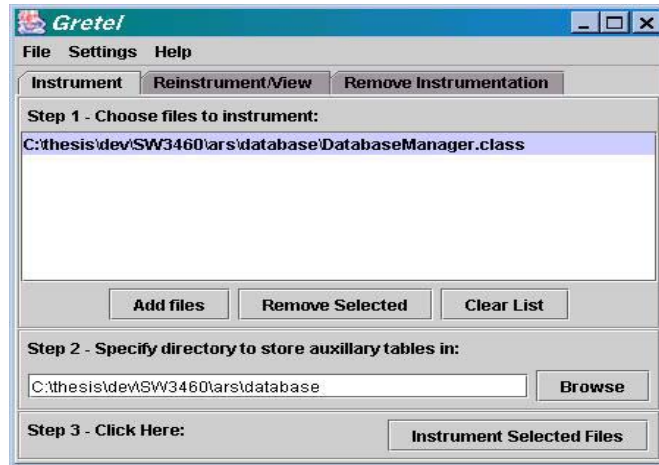


Figure 22. Instrumenting with Gretel

In the first ARS run, Gretel initially instruments the *DatabaseManager* class, which is a major ARS component. Figure 22 shows the specialized instrumentation GUI provided by Gretel. Next, the ARS will invoke a database operation, *query all flight data*, from the *DatabaseManager*. See Figure 23 below. Once invoked, Gretel will capture coverage results for this operation. The ARS is then exited and prepared for the next run.

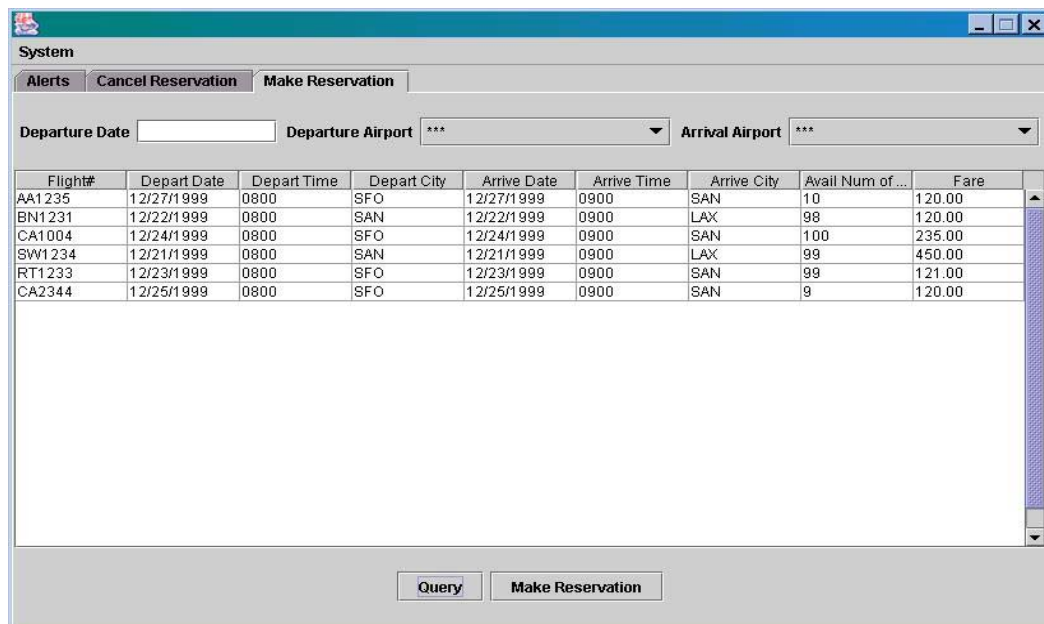


Figure 23. Querying All Flight Data in ARS

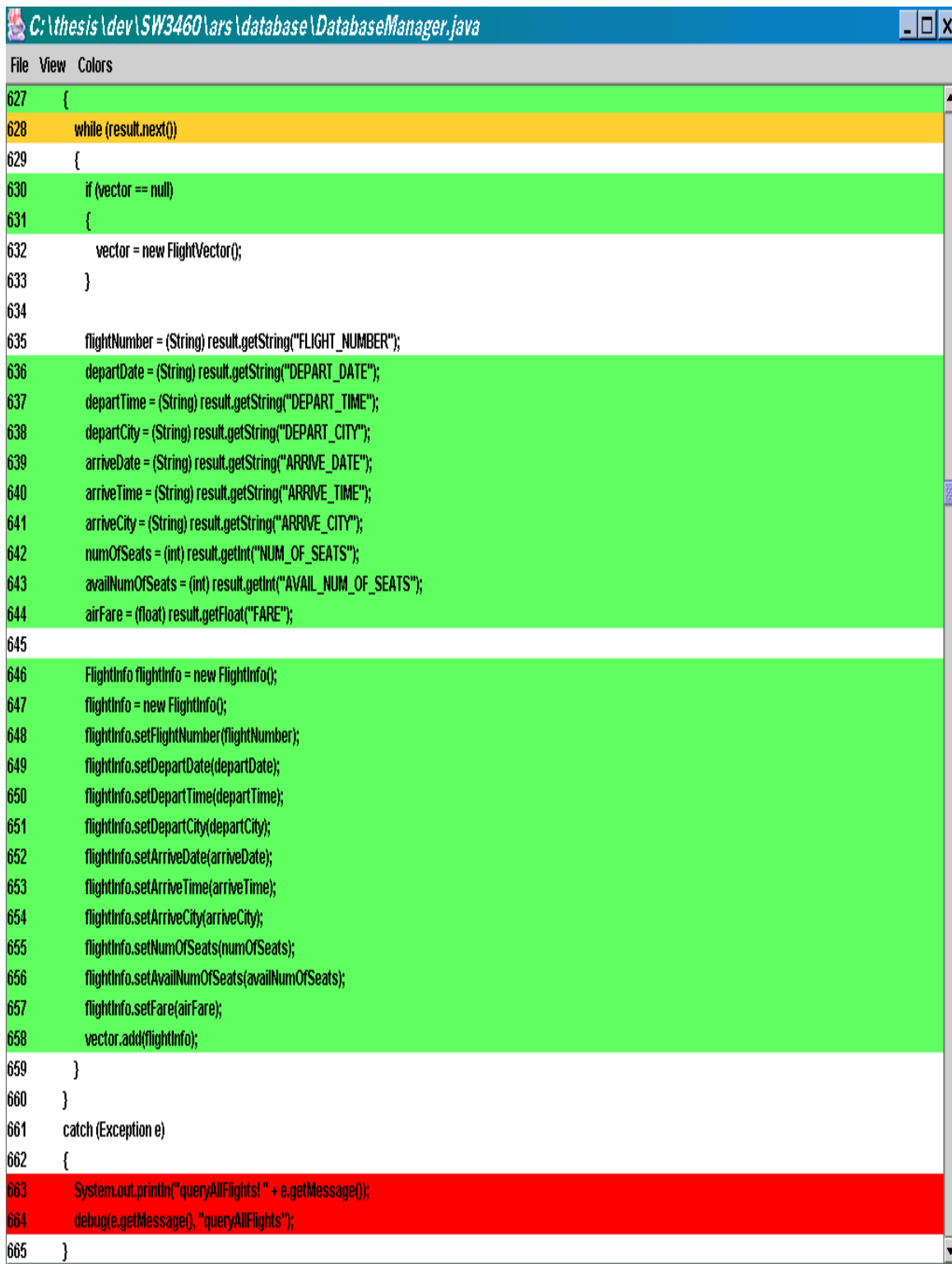
The second run will be somewhat similar to the first run, except that SIMPLE will inject a fault during execution. As in the first run, the ARS will invoke the same *DatabaseManager* database operation that was invoked. More importantly, the fault will exercise code uncovered by the results of the first test. Again, Gretel will capture coverage results for the ARS session.

4. Assessing SIMPLE Coverage

At the end of the second run, Gretel additionally incorporated coverage results from the first session. In other words, the coverage results from the second run actually represent a progressive coverage gathered from both sessions. Thus, the difference between both coverage sets will represent the increased coverage afforded by SIMPLE.

5. Results

In Figure 24, the Gretel coverage visualization tool shows the coverage results obtained from the first coverage run of the ARS. According to the visualization tool, lines 663 and 664 were never executed. Note in Figure 24 that lines 663 and 664 are colored red to indicate they were not executed. More specifically, these lines house the exception-handling code for this method.



```
C:\thesis\dev\SW3460\ars\database\DatabaseManager.java
File View Colors
627 {
628     while (result.next())
629     {
630         if (vector == null)
631         {
632             vector = new FlightVector();
633         }
634
635         flightNumber = (String) result.getString("FLIGHT_NUMBER");
636         departDate = (String) result.getString("DEPART_DATE");
637         departTime = (String) result.getString("DEPART_TIME");
638         departCity = (String) result.getString("DEPART_CITY");
639         arriveDate = (String) result.getString("ARRIVE_DATE");
640         arriveTime = (String) result.getString("ARRIVE_TIME");
641         arriveCity = (String) result.getString("ARRIVE_CITY");
642         numOfSeats = (int) result.getInt("NUM_OF_SEATS");
643         availNumOfSeats = (int) result.getInt("AVAIL_NUM_OF_SEATS");
644         airFare = (float) result.getFloat("FARE");
645
646         FlightInfo flightInfo = new FlightInfo();
647         flightInfo = new FlightInfo();
648         flightInfo.setFlightNumber(flightNumber);
649         flightInfo.setDepartDate(departDate);
650         flightInfo.setDepartTime(departTime);
651         flightInfo.setDepartCity(departCity);
652         flightInfo.setArriveDate(arriveDate);
653         flightInfo.setArriveTime(arriveTime);
654         flightInfo.setArriveCity(arriveCity);
655         flightInfo.setNumOfSeats(numOfSeats);
656         flightInfo.setAvailNumOfSeats(availNumOfSeats);
657         flightInfo.setFare(airFare);
658         vector.add(flightInfo);
659     }
660 }
661 catch (Exception e)
662 {
663     System.out.println("queryAllFlights! " + e.getMessage());
664     debug(e.getMessage(), "queryAllFlights");
665 }
```

Figure 24. Session 1 Coverage Results

In the second coverage run, SIMPLE specifically injected a fault that exercised previously untested portions of the code. In this case, SIMPLE nullified the *result* variable located in line 628 to evoke a *NullPointerException* in the method. The exception then triggered the code in lines 663 and 664. Appendix D-4 lists the contents of the fault configuration file used in this case study.

As expected, the coverage results, shown in Figure 25, reveals that the exception had indeed occurred and the exception-handling code was triggered. Note in Figure 25 that lines 663 and 664 are colored green to indicate that these lines were executed.



```
C:\thesis\dev\SW3460\ars\database\DatabaseManager.java
File View Colors
627 {
628     while (result.next())
629     {
630         if (vector == null)
631         {
632             vector = new FlightVector();
633         }
634
635         flightNumber = (String) result.getString("FLIGHT_NUMBER");
636         departDate = (String) result.getString("DEPART_DATE");
637         departTime = (String) result.getString("DEPART_TIME");
638         departCity = (String) result.getString("DEPART_CITY");
639         arriveDate = (String) result.getString("ARRIVE_DATE");
640         arriveTime = (String) result.getString("ARRIVE_TIME");
641         arriveCity = (String) result.getString("ARRIVE_CITY");
642         numOfSeats = (int) result.getInt("NUM_OF_SEATS");
643         availNumOfSeats = (int) result.getInt("AVAIL_NUM_OF_SEATS");
644         airFare = (float) result.getFloat("FARE");
645
646         FlightInfo flightInfo = new FlightInfo();
647         flightInfo = new FlightInfo();
648         flightInfo.setFlightNumber(flightNumber);
649         flightInfo.setDepartDate(departDate);
650         flightInfo.setDepartTime(departTime);
651         flightInfo.setDepartCity(departCity);
652         flightInfo.setArriveDate(arriveDate);
653         flightInfo.setArriveTime(arriveTime);
654         flightInfo.setArriveCity(arriveCity);
655         flightInfo.setNumOfSeats(numOfSeats);
656         flightInfo.setAvailNumOfSeats(availNumOfSeats);
657         flightInfo.setFare(airFare);
658         vector.add(flightInfo);
659     }
660 }
661 catch (Exception e)
662 {
663     System.out.println("queryAllFlights! " + e.getMessage());
664     debug(e.getMessage(), "queryAllFlights");
665 }
```

Figure 25. Session 2 Coverage Results

6. Discussion

Our final case study showed that SIMPLE improved test coverage. More importantly, the case study also showed how a coverage-analysis tool could be used to help direct and focus fault-injection testing. For instance, generated coverage reports can identify untested code within the application, in addition to untested exception-handling code. Hence, a coverage-analysis tool used in conjunction with a fault-injection test-harness would be a very valuable resource for testing system-critical software where maximal test coverage could be achieved via the abovementioned techniques.

Other test-coverage tools, such as *JCover*⁴¹, can provide accurate statistical coverage data that Gretel lacks. Such elaborate and sophisticated coverage data can be further analyzed to assess test adequacy, for example. For a more elaborate approach to assess test-adequacy using code-coverage metrics, see [10].

⁴¹ JCover is a code coverage analyzer for Java programs. See [63] for more details.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. CONCLUSION

Many of today's industries utilize Java as the developmental platform for their software. Much of Java's popularity is attributed to its support for multi-processing, concurrency, and rich APIs [71]. Thus, Java programming has been increasingly making its way into mission- and safety-critical systems. These systems require industrial strength software testing to ensure functional correctness. Unfortunately, weak testing can have catastrophic consequences, such as the well-known Therac-25 [5] and Ariane-5 systems mishaps [66].

An ideal test scheme for testing complex systems is to provide complete test coverage. However, it is impossible to investigate the entire input space of a system. As an example described in [72] a system with 40 binary inputs has an input space of 2^{40} or 10^{12} combinations. Thus, at a rate of one test per millisecond, it would take 35 years to test the system.

Traditionally, testers rely on pre-determined input distributions to test their software. However, even the most intricate input distribution set cannot guarantee that the software is correct. Part of the reason is that there could exist difficult-to-reach paths in the program. For example, exception handling requires certain conditions for execution.

In contrast to exhaustively checking for faults, faults can be injected into the SUT. The benefits of SWFI include fault acceleration, systematic testing and sensitivity analysis support, COTS testing, and improved test coverage.

Many SWFI tools exist today [1, 16, 17, 18, 21, 22, 25, 26, 28, 32]. Of these tools, many vary in their underlying SWFI technique, fault model, and usability. However, few SWFI tools exist today that are strictly Java-based. Thus, we proposed to develop our own SWFI tool, SIMPLE, so that others can fully appreciate the benefits SWFI offers in testing systems implemented in Java.

Ultimately, the associated risks with system applications lie in the hands of the owners, maintainers, and users. Thus, the owners and maintainers need to carefully test their systems for possible hazards and causal factors. Fortunately, existing SWFI tools,

such as SIMPLE, can be used to mitigate these risks. We have described our design process along with several case studies documenting the effectiveness of SIMPLE. However, we stress that SIMPLE is just a prototype and should not be looked upon as the “silver-bullet” against system anomalies.

Our case studies illustrated SIMPLE's potential for facilitating software testing. First of all, the first case study showed how SIMPLE verified the sensitivity of the test cases used for the CSMA/CD application. In other words, the system's test cases adequately responded to the faults that were injected by SIMPLE. Secondly, the second case study demonstrated how SIMPLE uncovered a weakness in the ARS system that the test cases failed to discover from previous testing. Specifically, a SIMPLE-emulated database error was not handled appropriately by the ARS system. Thus, causing inconsistency in the system's stored data. Lastly, the third case study confirmed that SIMPLE increased test-coverage. For instance, with the assistance of an open-source coverage tool (i.e., Gretel [60]), we proved that SIMPLE could tap into the alternate program paths and hard-to-reach source code such as exception handling.

IX. LIST OF REFERENCES

1. Schmid, M., Ghosh, A., Hill, F.: *Techniques for Evaluating the Robustness of Windows NT Software*, Reliable Software Technologies, 1999.
2. Johnson, D. W.: *Java Technology in the Department of Defense*, JavaOne Conference, 1999.
3. Voas, J.; McGraw, G.: *Software Fault Injection; Inoculating Programs Against Errors*, John Wiley & Sons, 1997.
4. Slabodkin, G.: *Software Glitches Leave Navy Ship Dead In The Water*, Government Computer News, <http://www.gcn.com>, July 13, 1998.
5. Leveson, N.; Turner, S.: *An Investigation of the Therac-25 Accidents*, IEEE Computer, 1993, pp. 18-41.
6. Voas, J.; Kassab, L.; Voas, L.: *A "Crystal Ball" for Software Liability*, IEEE, 1997.
7. Toress-Pomales, W.: *Software Fault Tolerance: A Tutorial*, NASA/TM-2000-2106, October 2000.
8. McDougall, P.: *Microsoft Will Release APIs To Satisfy Mandates*, InformationWeek, pp.20, August 12, 2002.
9. Shelton, C. P.; Koopman, P.; DeVale, K.: *Robustness Testing of the Microsoft Win32 API*, International Conference on Dependable Systems and Networks, IEEE, 2000.
10. Bieman, J. M.; Dreilinger, D.; Lin, Lijun: *Using Fault Injection to Increase Software Test Coverage*, Proceedings of International Symposium on Software Reliability, ISSRE, 1996.
11. Voas, J.: *Discovering Unknown Software Output Modes and Missing System Hazards*, lecture at Naval Postgraduate School, April 26, 2002.
12. Chiba, S.: *Load-time Structural Reflection in Java*, ECOOP 2000, Object-Oriented Programming, LNCS 1850, Springer Verlag, pp. 313-336, 2000.
13. Welch, I.; Stroud, R. J.: *Kava – A Reflective Java based on Bytecode Rewriting*, Proceedings of USENIX Conference on Object-Oriented Technology, 2001.
14. Martins, E.; Rosa, A.: *A Fault Injection Approach based on Reflective Programming*, Proceedings of International Conference on Dependable Systems and Networks (DSN), pg. 409, 2000.
15. Haddox, J.; Kapfhammer, G.; Michael, Ph.D., C.C.; Schatz, M.: *Testing Commercial-Off-The-Shelf With Software Wrappers*, Cigital, Inc.
16. Carreira, J.; Madeira, H.; Gabriel, M.: *Xception: Software Fault Injection and Monitoring in Processor Functional Units*, IEEE Transactions on Software Engineering, Vol.24, No.2, February 1998.
17. Dawson, S.; Jahanian, F.; Mitton, T.: *ORCHESTRA: A Fault Injection Environment for Distributed Systems*, 26th International Symposium on Fault Tolerant Computing (FTCS), 1996.
18. Stott, D.T.; Kalbarczyk, Z.; Iyer, R.K.: *Using NFTAPE for Rapid Development of Automated Fault Injection Experiments*, 1999.
19. S. Ghosh, A.; Mathur, A. P.: *Interface Mutation*, Journal of Testing, Verification and Reliability, pp 227-247, Volume 11, Issue 4, December 2001.
20. Hseuh, M.; Sai, T.K.; Iyer, R.K.: *Fault Injection Techniques and Tools.*, IEEE, 1997.
21. Aidemark, J.; Vinter, J.; Folkesson, P.; Karlsson, J.: *GOOFI: Generic Object-Oriented Fault Injection Tool*. The International Conference on Dependable Systems and Networks (DSN), July 2001.
22. Han, S.; Rosenberg, H.A.; Shin, K.G.: *DOCTOR: An Integrated Software Fault Injection Environment*, Proc. of IEEE International Computer Performance and Dependability Symposium, April 1995.
23. W. Du, W.; Mathur, A.P.: *Vulnerability Testing of Software System Using Fault Injection*. Technical Report Coast TR98 -02, Department of Computer Science, Purdue University, 1998.
24. Ghosh, A.K, Voas, J.: *Inoculating Software For Survivability*, Communications of the ACM, Volume 42, Issue 7, pp. 38-44, July 1999.
25. Stott, D.T.; Floering, B.; Kalbarczyk, Z.; Ravishankar, K. I.: *A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors*, Proceedings of the 4th International Computer Performance and Dependability Symposium, IEEE, 1998.

26. Kao, W.I., Iyer, R.K., Tang, D.: *FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults*, IEEE, 1993.
27. Binder, R.V.: *Testing Object-Oriented Systems Models, Patterns, and Tools*, Addison Wesley, 1999.
28. Kanawati, G.; N. Kanawati, N.; Abraham, J.: *FERRARI: A Flexible Software-Based Fault and Error Injection System*, IEEE Transactions on Computers, 44 (2), Feb. 1995.
29. Michael, J. B.: *Requirements Analysis: Generic and Derived Safety Requirements*, lecture for SW4582: *Systems Software Safety* course at Naval Postgraduate School, Oct 2001.
30. Voas, J.; Miller, K.W.: *Software Testability: The New Verification*, IEEE, May 1995.
31. Voas, J.: *Testability Based Assertion Injection for Software Debugging*, Proceedings of Second International Workshop of Automated and Algorithmic Debugging (AADEBUG), May 1995.
32. Tsai, T.K.; Iyer, R.K.: *Measuring Fault Tolerance with the FTape Fault Injection Tool*, Center for Reliable and High Performance Computing, Coordinated Sciences Laboratory, 1994.
33. Ghosh, S.; Mathur, A.P.; Horgan, J.R.; Li, J.J.; Wond, E.W.: *Software Fault Injection Testing on a Distributed System – A Case Study*, First International Software Quality Week Europe, Brussels, Belgium, November 4-7 1997.
34. Tsai, T.K.; Iyer, R.K.: *An Approach to Benchmarking of Fault-Tolerant Commercial Systems*, Proceedings of the Second Annual IEEE International Computer Performance and Dependability Symposium, IEEE, p. 204-213, 1995.
35. Carney, D.: *Java Perks Up Federal Applications*, Federal Computer Weekly, June 21, 1999.
36. World Wide Web Consortium (W3C). "Extensible Markup Language (XML)." [<http://www.w3.org/XML/>]. September 2002.
37. Chiba, Shigeru. "Javassist Home Page." [<http://www.csg.is.titech.ac.jp/~chiba/javassist/>]. October 2002.
38. Palo Alto Research Center Incorporated. "Aspectj.org." [<http://aspectj.org/servlets/AJSite>]. 2002.
39. Sun Microsystems, Inc. "Java Platform Debugger Architecture." [<http://java.sun.com/j2se/1.4/docs/guide/jpda/>]. 2002.
40. Compaq. "Compaq JTrek." [<http://www.compaq.com/java/download/jtrek/>]. December 2002.
41. The Apache Software Foundation. "Xerces2 Java Parser." [<http://xml.apache.org/xerces2-j/index.html>]. 2002.
42. Louderback, Jim. "Any PC Can Fall Victim To The Heisenberg Principle." [<http://www.ncns.com/news/2/heisenberg.html>]. June 1997.
43. Webopedia. "Definition of Overhead." [<http://www.webopedia.com/TERM/o/overhead.html>]. 2002.
44. Sun Microsystems, Inc. "Java Message Service API." [<http://java.sun.com/products/jms/>]. 2002.
45. Thornhus, R.: *Software Fault Injection Testing*, Master of Science Thesis in Electronic System Design, Ericsson Telecom, February 2000.
46. Webgain, Inc. "Java Compiler Compiler (JavaCC) - The Java Parser Generator." [http://www.webgain.com/products/java_cc/]. 2002.
47. Pazandak, P., Wells, D., "Probemeister: Distributed Runtime Software Instrumentation." [<http://joint.org/use2002/sub/pazandak-ProbeMeister.pdf>]. 2002.
48. Cigital. "Case Study II: Finding Defects Earlier Saves Big \$\$\$." [<http://www.cigital.com/solutions/roi-cs2.html>]. 2002.
49. Sadiku, M. and Ilyas, M., *Simulation of Local Area Networks*, Boca Raton, Florida. CRC Press, 1994, pp. 112-133.
50. Hightower, R.; Lesiecki, N.: *Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus*, John Wiley and Sons, 2001.
51. Beck, K.: *Extreme Programming Explained*, Addison-Wesley, 2000.
52. Choi, B.: *Test Adequacy Measurement Using a Combination of Criteria*, International Journal of Reliability, Quality and Safety Engineering, Vol. 7, No. 3, 2000, pp. 191-203.
53. Weyuker, E. J.: *Axiomatizing Software Test Data Adequacy*, IEEE TSE, Vol. SE_12, No.12, 1986, pp. 1128-1138.
54. Zhu, H.; Hall, P.; May, J.: *Software Unit Test Coverage and Adequacy*, ACM Computing Surveys, Vol. 29, No. 4, December 1997.

55. Johnson, J.: *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan Kaufmann Publishers, March 2000.
56. Holub, A: *Taming Java Threads*, Apress, 2000.
57. Hyde, P.: *Java Thread Programming*, Sams, 1999.
58. Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1999.
59. Pavlopoulou, C.; Young, M.: *Residual Test Coverage Monitoring*, Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press, 1999, pp. 227 – 284.
60. Young, M., Howells, C. (University of Oregon), "Gretel: An Open Source Residual Test Coverage Tool." [<http://www.cs.uoregon.edu/research/perpetual/Software/Gretel/>]. June 2002.
61. SourceForge.net. "Hansel 0.02." [<http://hansel.sourceforge.net/>]. 2002.
62. SourceForge.net. "GretAnt." [<http://gretant.sourceforge.net/>]. 2002.
63. Codework. "JCover: Java Code Coverage Testing and Analysis." [<http://www.codework.com/JCover/product.html>]. 2002.
64. Miller, S. K.: *Aspect-Oriented Programming Takes Aim at Software Complexity*, Technology News, Vol.34, No.4, April 2001, pp. 18-21.
65. Brooks, F.: *The Mythical Man-Month*, Addison-Wesley, 1995.
66. Nuseibeh, B.: *Ariane 5: Who Dunnit?*, IEEE, 1997 , Report by the Inquiry Board, *Ariane 5 Flight 501 Failure*, <http://java.sun.com/people/jag/Ariane5.html>, 1996.
67. SourceForge.net. "FIDe: Fault Injection via Debugging." [<http://fide.sourceforge.net/>]. 2002.
68. SourceForge.net. "Linux Fault Injection Test Harness." [<http://fault-injection.sourceforge.net/>]. 2002.
69. SourceForge.net. "JPWrite." [<http://jpwrite.sourceforge.net/>]. 2002.
70. Sun Microsystems, Inc. "JDBC Data Access API." [<http://java.sun.com/products/jdbc/>]. November 2002.
71. Paula, G.: *Java Catches on for Manufacturing*, The American Society of Mechanical Engineers, December 1997.
72. Storey, N.: *Safety Critical Computer Systems*, 2nd Ed, Prentice Hall, 1996.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A – SIMPLE UML DIAGRAMS

This appendix contains UML diagrams that describe some of the design aspects of SIMPLE.

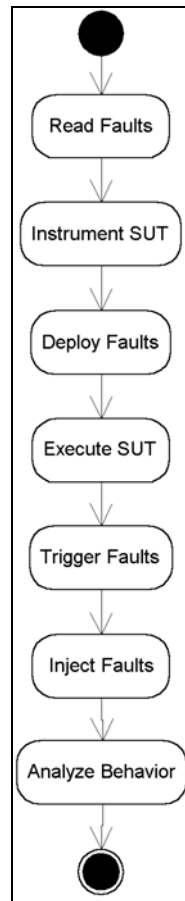


Figure A-1. SIMPLE Activity Diagram

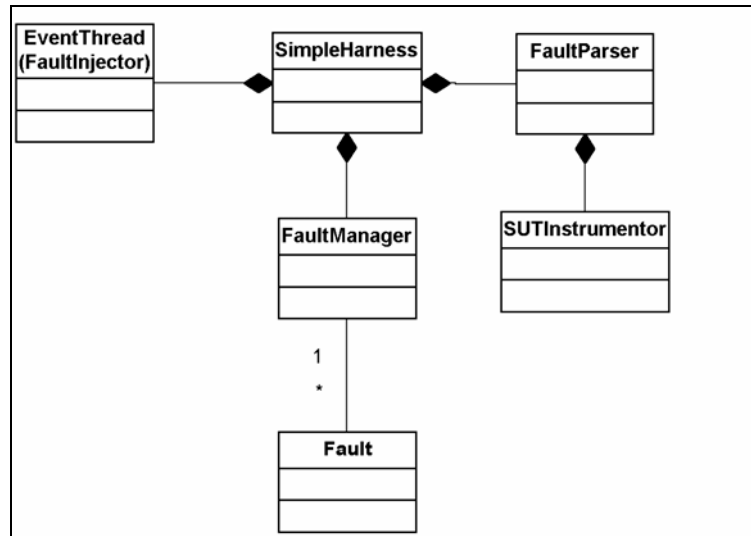


Figure A- 2. High-level Class Diagram of SIMPLE Components

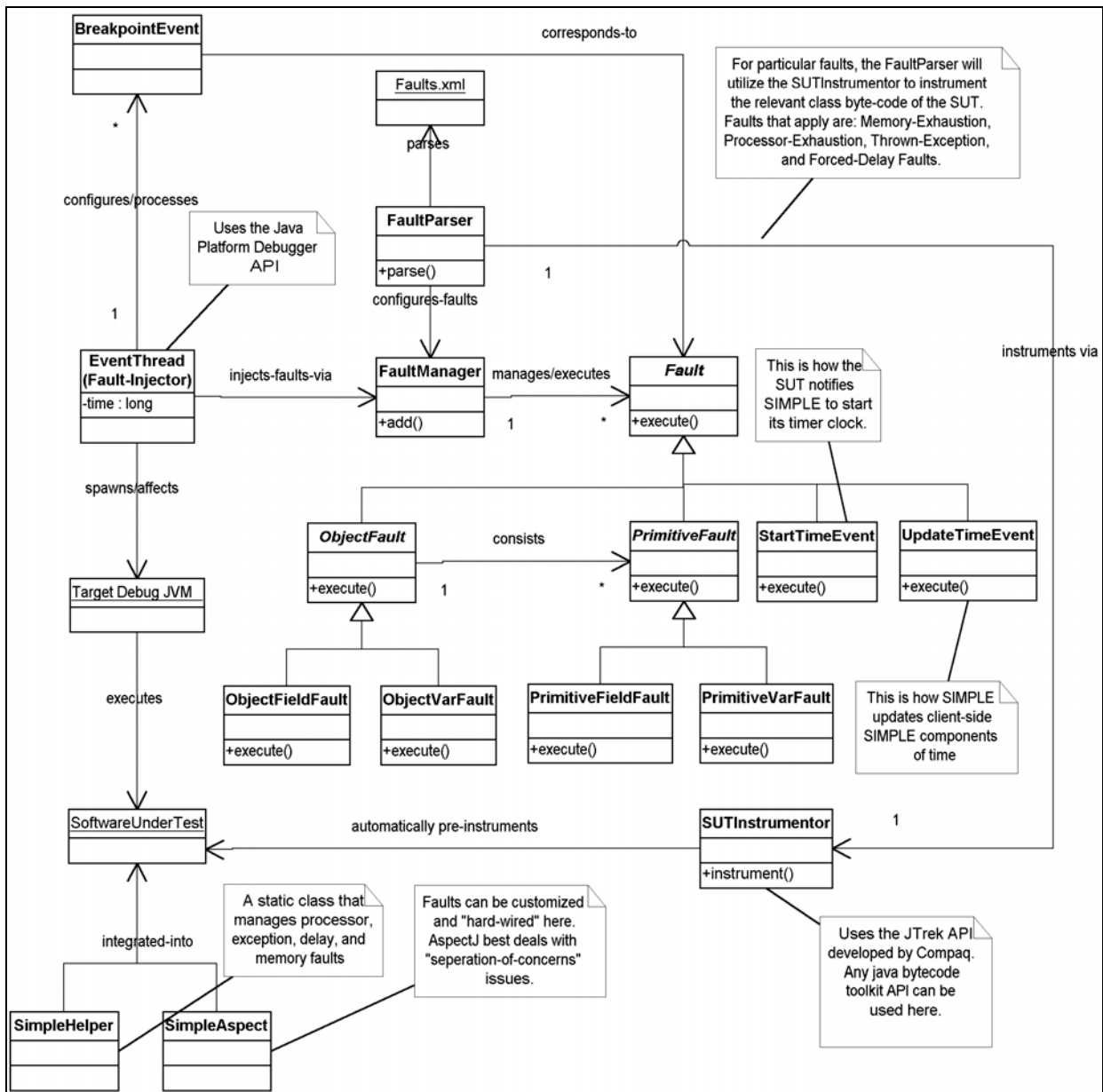


Figure A-3. Detailed Class Diagram of SIMPLE Components

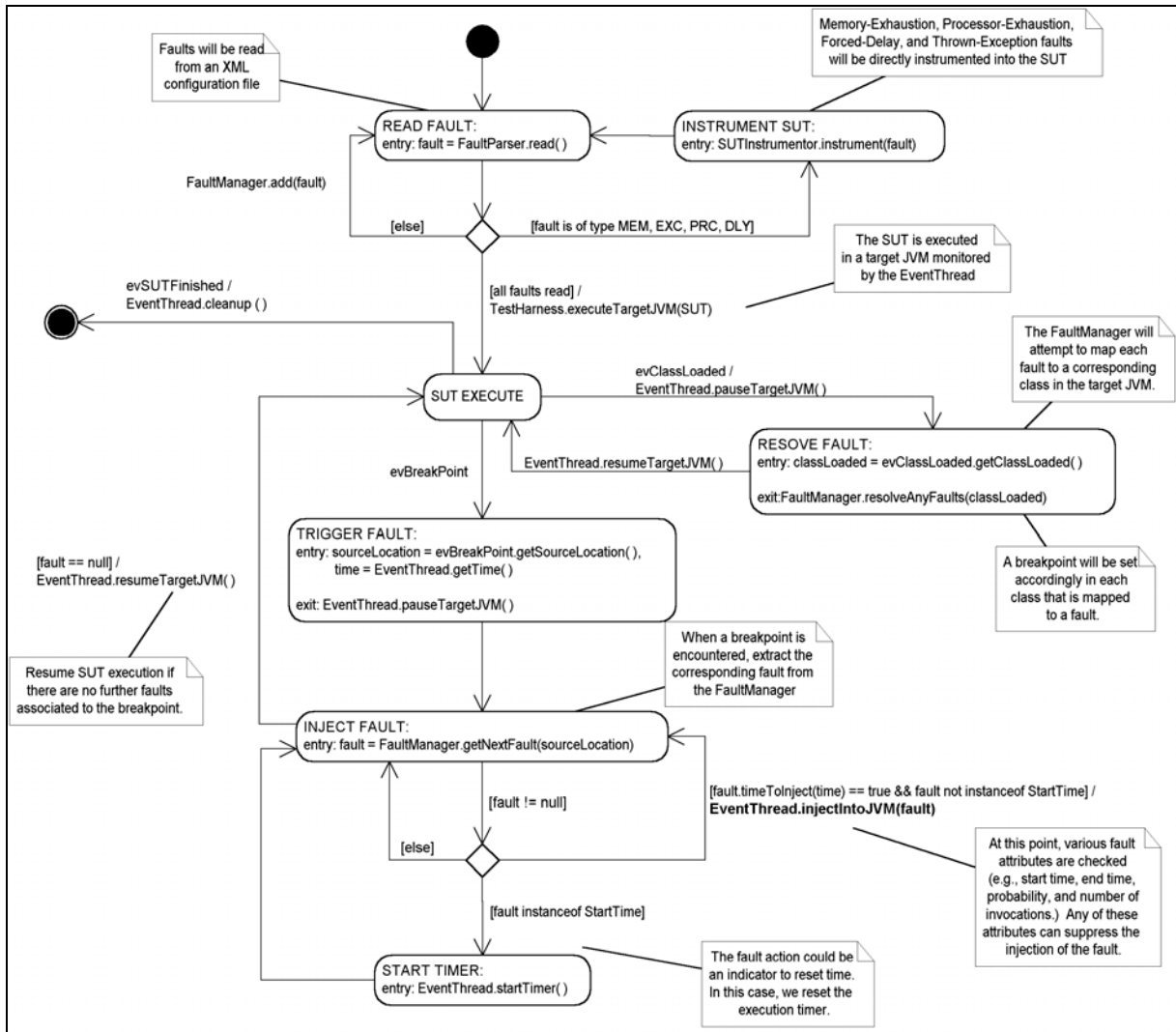


Figure A-4. General State Diagram of SIMPLE Processes

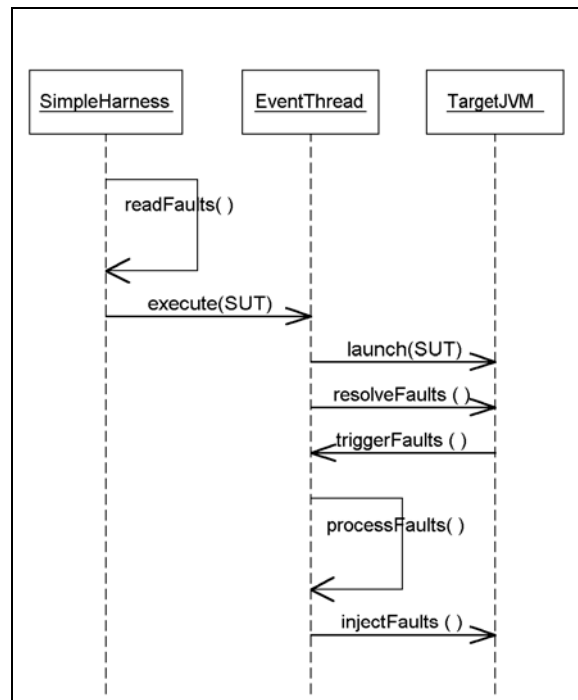


Figure A-5. High-level Sequence Diagram of SIMPLE Process

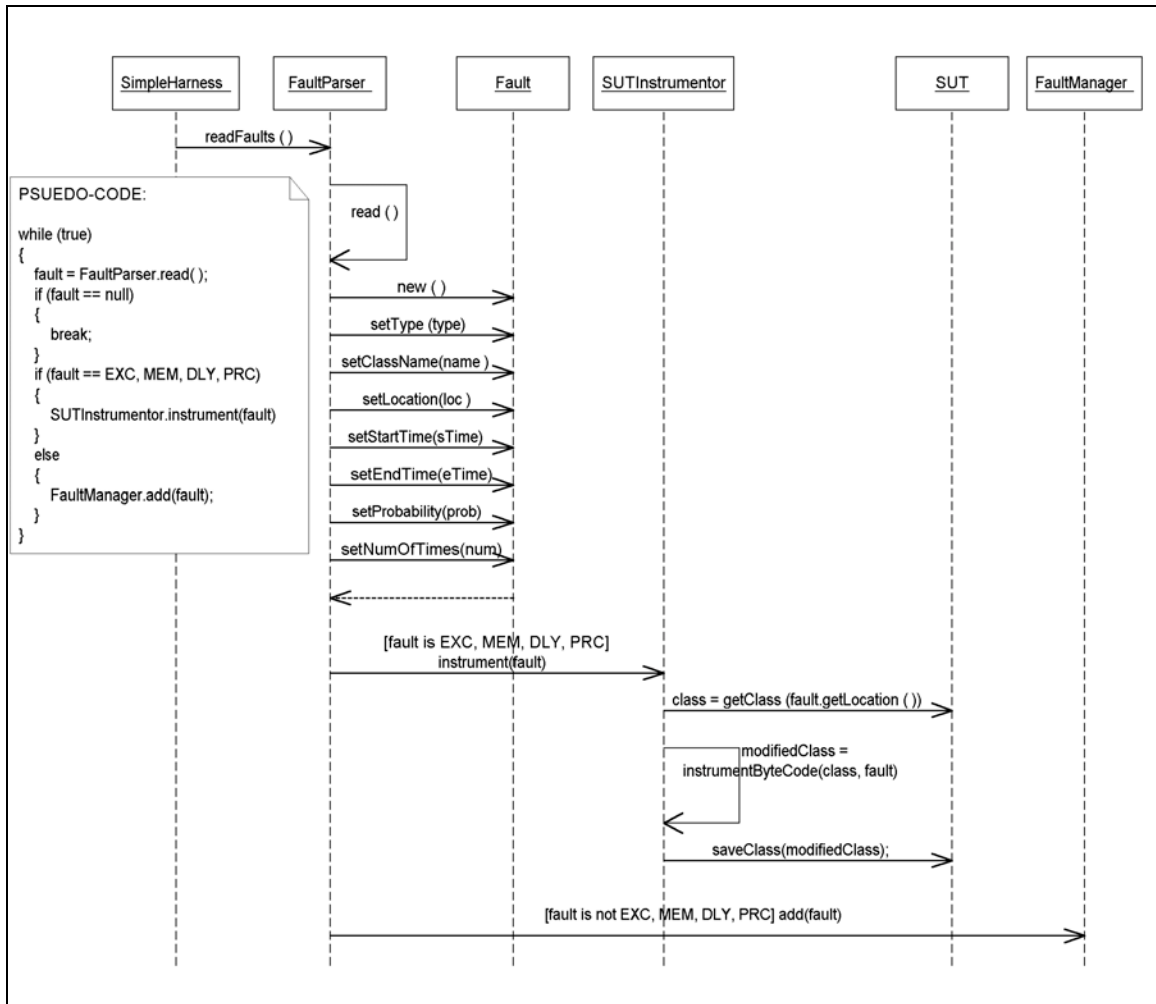


Figure A-6. Sequence Diagrams of Faults Being Parsed

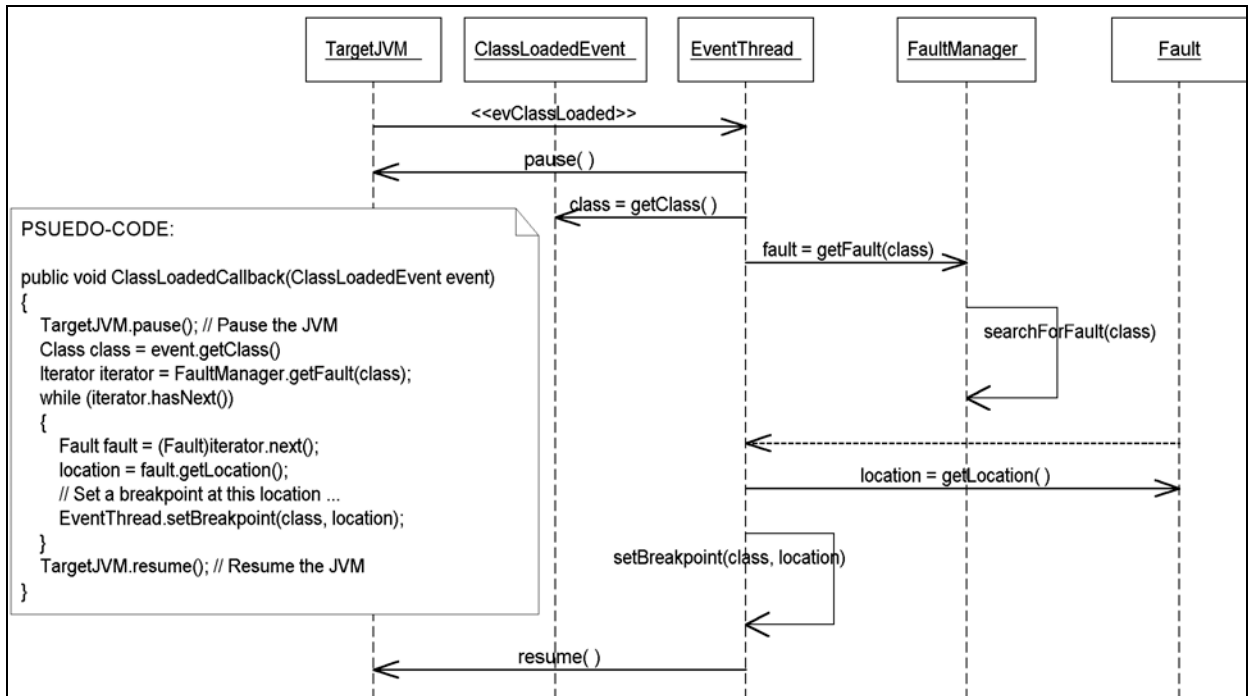


Figure A-7. Sequence Diagram of Classes Being Prepared

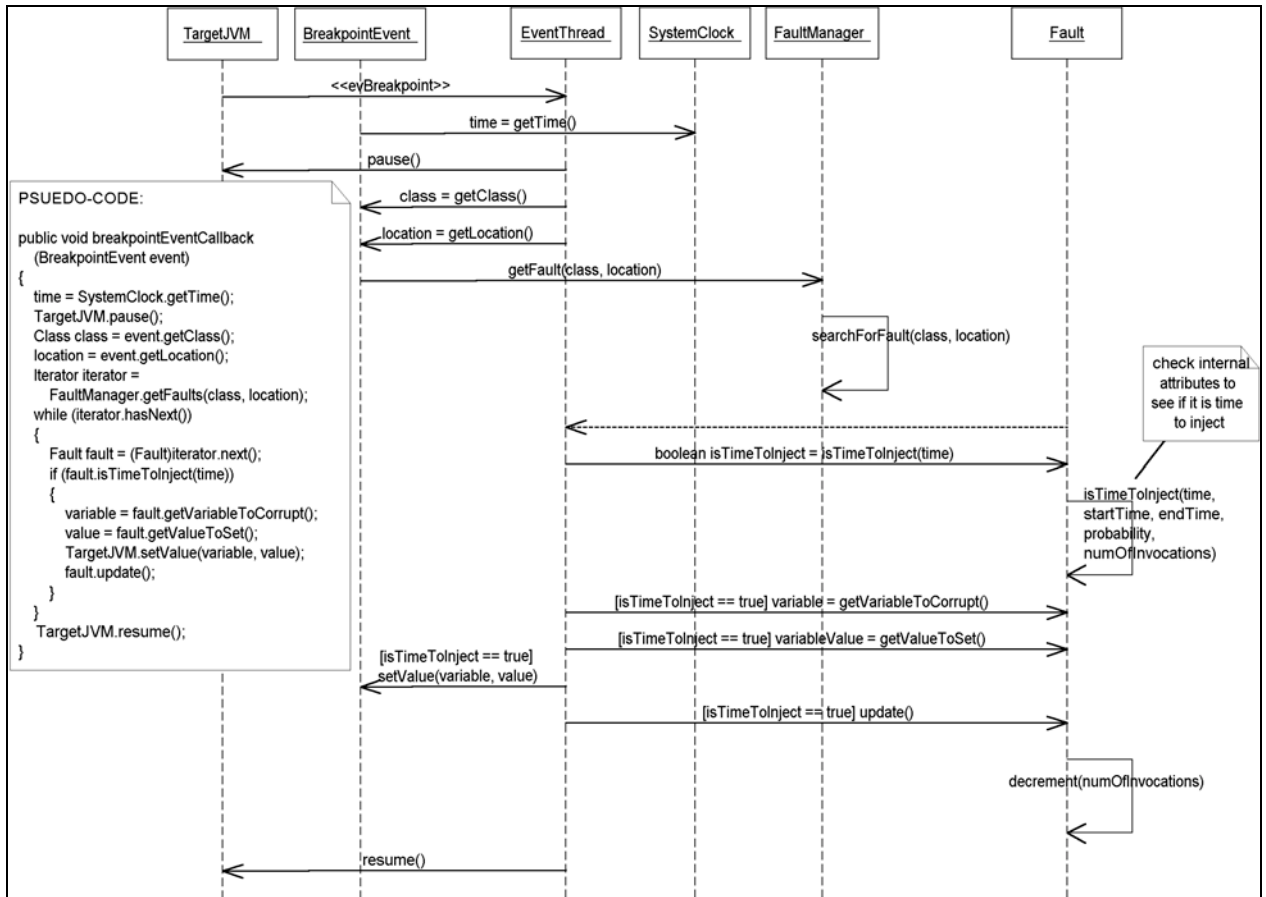


Figure A-8. Sequence Diagram of Fault Triggers Being Served

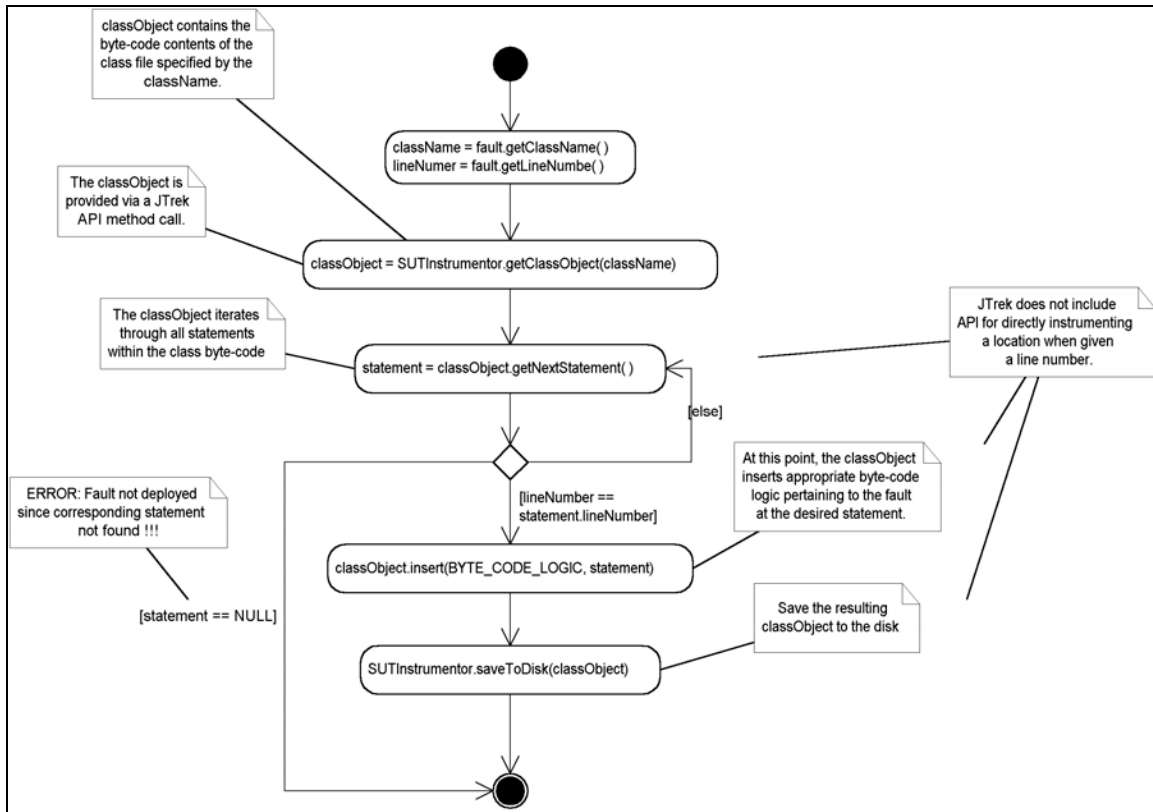


Figure A- 9. State Diagram of SUT Instrumentation

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B – FAULT SPECIFICATION GRAMMAR

This appendix contains the fault specification grammar. More specifically, the XML data elements of the fault configuration file will be described here in grammar form. Where necessary, annotated descriptions are provided. Currently, the tester provides this fault configuration file to SIMPLE, but future work would entail the implementation of a fault deployment support tool that automatically generates the fault configuration file.

Rule		Rule Expansion	Description
<i>TopFaultNode</i>	::=	“<FaultConfig>” <i>FaultNode</i> * “</FaultConfig>”	This is the <i>entry</i> rule for defining the fault configuration file.
<i>FaultNode</i>	::=	“<Fault “ [<i>FaultAttribute</i>]* “>” <i>FaultType</i> * “</Fault>”	This rule defines a fault in SIMPLE where fault attributes are set. The type of the fault is provided by the <i>FaultType</i> rule.
<i>FaultAttribute</i>	::=	“className=”string “””	Specifies the class where the fault is to occur.
		“lineNo=”integer “””	Specifies the line number of the class where the fault is to occur
		“numOfInvoc=”integer “””	Specifies the number of times the fault is to occur.
		“prob=”float“””	Specifies the probability

			that the fault is to occur.
		<code>“startTime=”integer“” </code>	Specifies the starting time that the fault is to occur.
		<code>“endTime=”integer“” </code>	Specifies the ending time that the fault is to occur.
		<code>“activateAt=”string”.”integer“” </code>	Specifies a class location at which to activate the fault.
		<code>“deactivateAt=”string”.”integer“” </code>	Specifies a class location at which to deactivate the fault.
		<code>“varName=”string“” </code>	Specifies the variable to be corrupted by the fault.
		<code>“valToSet=”float / integer“” </code>	Specifies the corruption value to apply to the variable.
		<code>“setToNull=”boolean“” </code>	Specifies whether to apply the <i>null</i> value to the variable.
		<code>“arg=”integer“” </code>	Specifies a generic argument to be used by the Processor, Memory, Delay, and Exception faults.
		<code>“enable=”boolean “”</code>	Specifies whether the fault is to be initially enabled or not.
<i>FaultType</i>	<code>::=</code>	<code>“<PrimField “[FaultTypeAttribute*]”</code>	Specifies that the fault

		<i>FaultType</i> *	involves the corruption of a class field that is a primitive type.
		“<PrimLocal “[<i>FaultAttribute</i> *” “/>”	Specifies that the fault involves the corruption of a local method variable that is a primitive type (e.g., integer, float, double, etc).
		“<ObjField “[<i>FaultAttribute</i> *” <i>FaultType</i> *	Specifies that the fault involves the corruption of a class field that is a object type.
		“<ObjLocal “[<i>FaultAttribute</i> *” <i>FaultType</i> *	Specifies that the fault involves the corruption of a local method variable that is an object type.
		“<Processor “[<i>FaultAttribute</i> *” “/>”	Specifies a processor fault.
		“<Memory “[<i>FaultAttribute</i> *” “/>”	Specifies a memory fault.
		“<Delay “[<i>FaultAttribute</i> *” “/>”	Specifies a delay fault.
		“<Exception “[<i>FaultAttribute</i> *” “/>”	Specifies an exception fault.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C – CASE STUDY UML DIAGRAMS

This appendix contains UML diagrams that supplement the case studies discussed in Chapter VII.

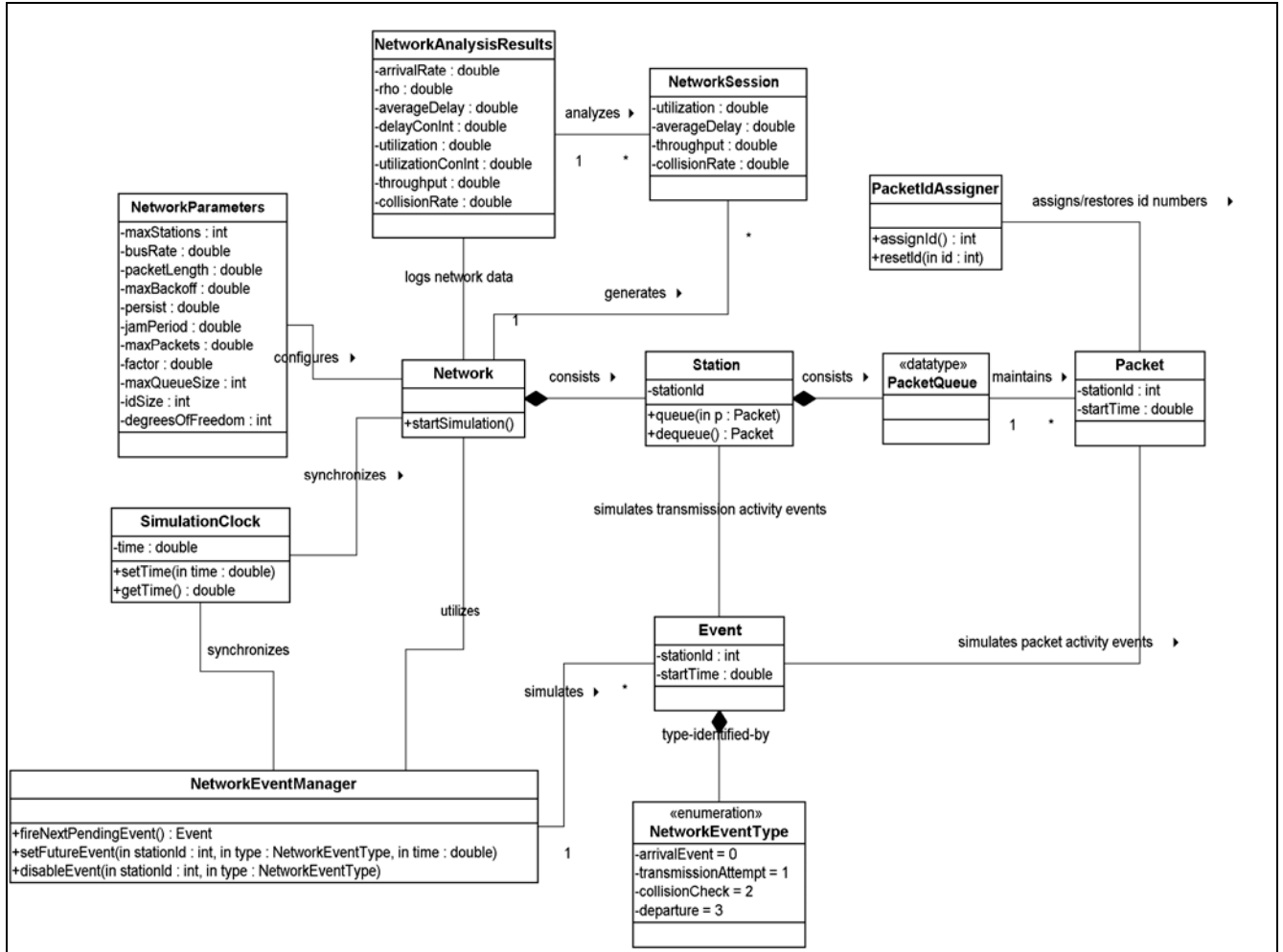


Figure C- 1. Class Diagram for the CSMA/CD Simulation Software

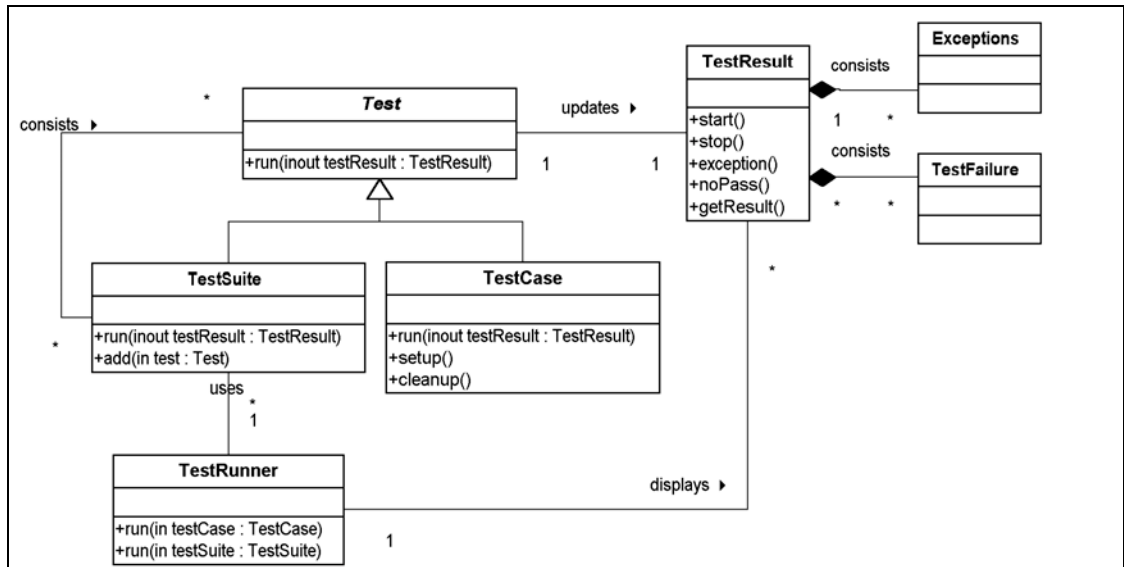


Figure C- 2. JUnit Framework Class Diagram

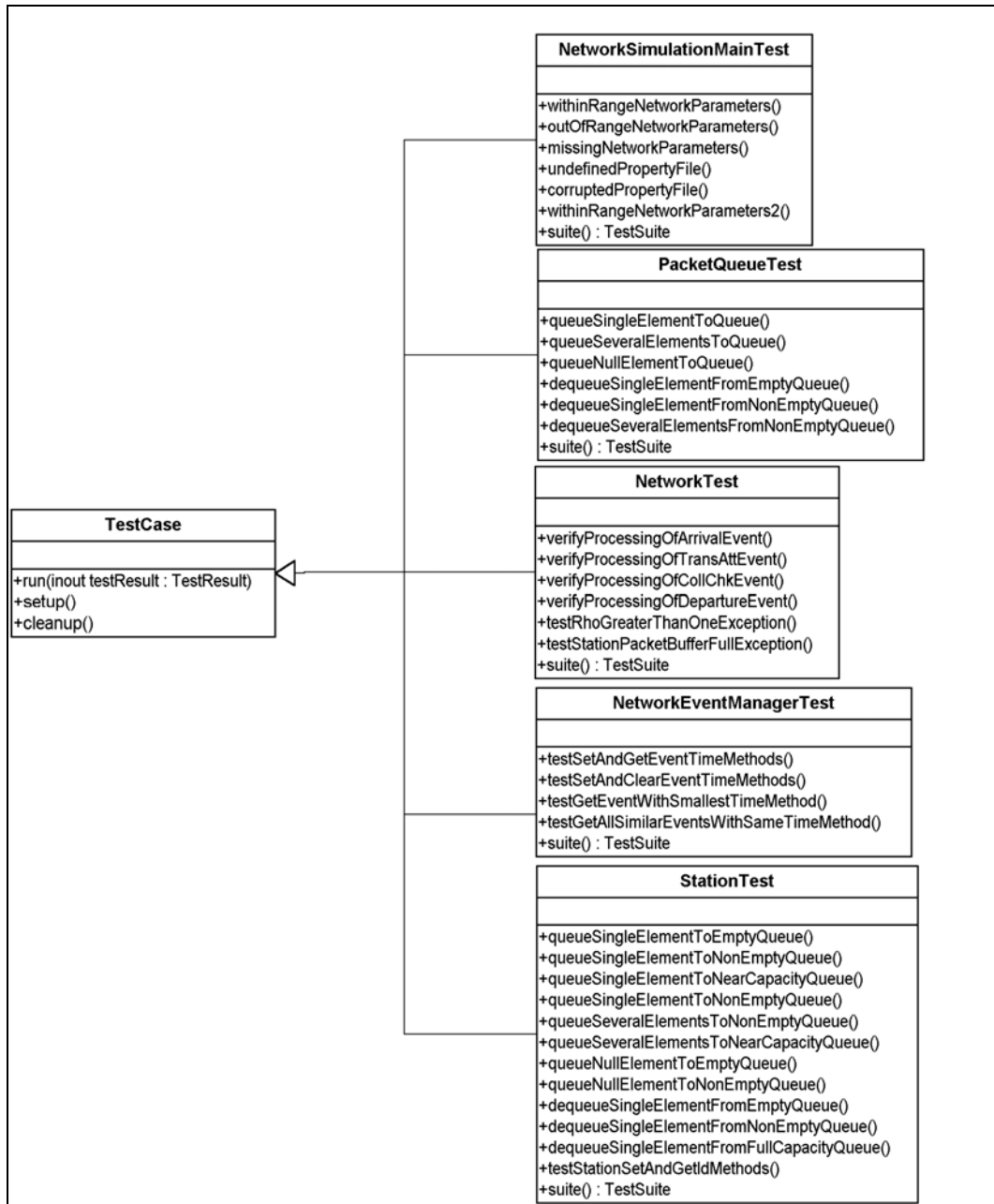
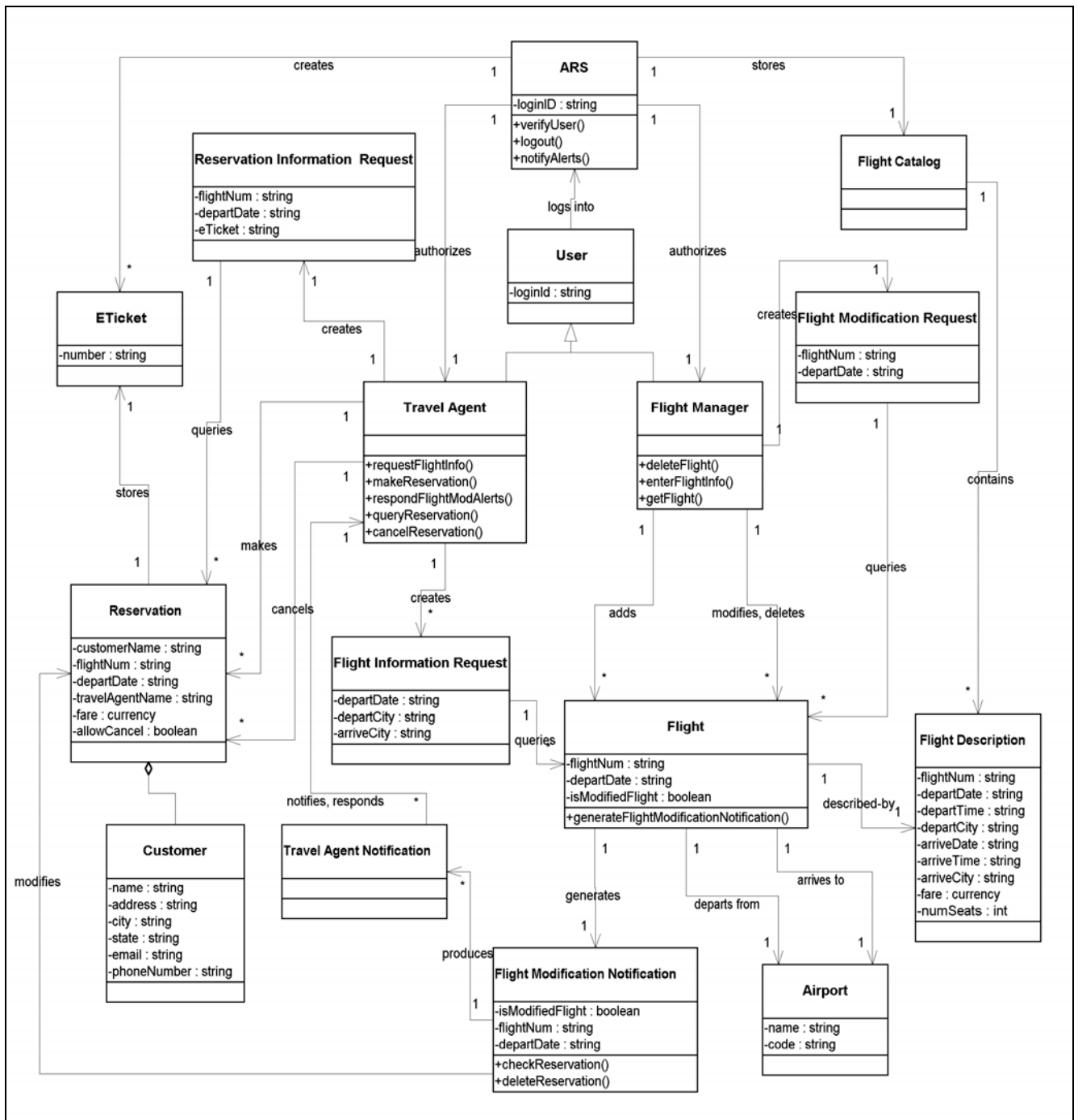


Figure C- 3. Test Suite Class Diagrams



APPENDIX D – CASE STUDY FAULT CONFIGURATION FILES

This appendix contains the SIMPLE fault configurations that were used in the case studies described in Chapter VII.

D-1 CSMA/CD UNIT-TEST FAULT CONFIGURATION FILE (CASE STUDY I)

```
<!--
    Fault configuration for the "SimpleTest" TestSuite Case-Study #1
        developed by Chris Acantilado and Neil Acantilado
-->

<FaultConfig>

    <!--
        NetworkSimulationMainTest
    -->

    <!-- Test Case 1: withinRangeNetworkParameters-->
    <Fault class="csma.app.NetworkSimulationMainTest" lineNo="58"
        numOfInvoc="1">
        <PrimField varName="numOfRuns" valToSet="-999"/>
    </Fault>

    <!-- Test Case 2: outOfRangeNetworkParameters-->
    <Fault class="csma.app.NetworkSimulationMainTest" lineNo="92"
        numOfInvoc="1">
        <PrimField varName="packetLength" valToSet="-999"/>
    </Fault>

    <!-- Test Case 3: queueSingleElementToQueue -->
    <Fault class="csma.app.NetworkSimulationMainTest" lineNo="128"
        numOfInvoc="1">
        <PrimField varName="maxPackets" valToSet="-999"/>
    </Fault>

    <!--
        PacketQueueTest
    -->

    <!-- Test Case 1: queueSingleElementToQueue -->
    <Fault class="csma.client.PacketQueueTest" lineNo="44" numOfInvoc="1">
        <ObjLocal varName="packet" setToNull="true" />
    </Fault>

    <!-- Test Case 2: queueSeveralElementsToQueue -->
    <Fault class="csma.client.PacketQueueTest" lineNo="63" numOfInvoc="1" >
        <PrimLocal varName="i" valToSet="5"/>
    </Fault>

    <!-- Test Case 5: dequeueSingleElementFromNonEmptyQueue -->
    <Fault class="csma.client.PacketQueueTest" lineNo="121" numOfInvoc="1" >
        <ObjLocal varName="packet">
            <PrimField varName="packetId" valToSet="0"/>
            <PrimField varName="startTime" valToSet="0"/>
        </ObjLocal>
    </Fault>

    <!-- Test Case 6: dequeueSeveralElementsFromNonEmptyQueue -->
    <Fault class="csma.client.PacketQueueTest" lineNo="145" numOfInvoc="1" >
        <ObjLocal varName="packet">
            <PrimField varName="packetId" valToSet="8"/>
            <PrimField varName="startTime" valToSet="8"/>
        </ObjLocal>
    </Fault>

    <!--
        StationTest
```

```

-->

<!-- Test Case 4: queueSingleElementToFullCapacityQueue -->
<Fault class="csma.client.StationTest" lineNo="104" numOfInvoc="1" >
  <ObjLocal varName="station">
    <PrimField varName="maxQueueSize" valToSet="9999"/>
  </ObjLocal>
</Fault>

<!-- Test Case 5: queueSeveralElementsToNonEmptyQueue BEGIN -->
<Fault class="csma.client.StationTest" lineNo="123" numOfInvoc="1" >
  <ObjLocal varName="station">
    <PrimField varName="maxQueueSize" valToSet="10000"/>
  </ObjLocal>
</Fault>

<Fault class="csma.client.StationTest" lineNo="125" numOfInvoc="1">
  <PrimLocal varName="i" valToSet="-5"/>
</Fault>

<!-- Test Case 6: queueSeveralElementsToNearCapacityQueue -->
<Fault class="csma.client.StationTest" lineNo="140" numOfInvoc="1" >
  <ObjLocal varName="station">
    <PrimField varName="maxQueueSize" valToSet="20000"/>
  </ObjLocal>
</Fault>

<!-- Test Case 10: dequeueSingleElementFromNonEmptyQueue -->
<Fault class="csma.client.StationTest" lineNo="226" numOfInvoc="1" >
  <ObjLocal varName="packet">
    <PrimField varName="packetId" valToSet="0"/>
    <PrimField varName="startTime" valToSet="0"/>
  </ObjLocal>
</Fault>

<!-- Test Case 12: testStationSetAndGetIdMethods -->
<Fault class="csma.client.StationTest" lineNo="267" numOfInvoc="1" >
  <ObjLocal varName="station">
    <PrimField varName="stationId" valToSet="0"/>
  </ObjLocal>
</Fault>

<!--
  NetworkEventManagerTest
-->

<!-- Test Case 1: testSetAndGetEventTimeMethods -->
<Fault class="csma.event.NetworkEventManagerTest" lineNo="53"
  numOfInvoc="1" >
  <PrimLocal varName="expectedTime" valToSet="-999"/>
</Fault>

<!-- Test Case 2: testSetAndClearEventTimeMethods -->
<Fault class="csma.event.NetworkEventManagerTest" lineNo="129"
  numOfInvoc="1" >
  <PrimLocal varName="transAttTime" valToSet="-999"/>
</Fault>

<!-- Test Case 3: testGetEventWithSmallestTimeMethod -->
<Fault class="csma.event.NetworkEventManagerTest" lineNo="212"
  numOfInvoc="1" >
  <PrimLocal varName="eventTime" valToSet="-999"/>
</Fault>

```

```

<!--
    NetworkTest
-->

<!-- Test Case 1: verifyProcessingOfArrivalEvent -->
    <Fault class="csma.network.Network" lineNo="642" numOfInvoc="1"
        activateAt="csma.network.NetworkTest:47"
        deactivateAt="csma.network.NetworkTest:53">
        <PrimField varName="isArrivalEventProcessed" valToSet="false"/>
    </Fault>

<!-- Test Case 2: verifyProcessingOfTransAttEvent -->
    <Fault class="csma.network.Network" lineNo="651" numOfInvoc="1"
        activateAt="csma.network.NetworkTest:72"
        deactivateAt="csma.network.NetworkTest:78">
        <PrimField varName="isTransmissionAttemptEventProcessed"
valToSet="false"/>
    </Fault>

<!-- Test Case 3: verifyProcessingOfCollChkEvent -->
    <Fault class="csma.network.Network" lineNo="660" numOfInvoc="1"
        activateAt="csma.network.NetworkTest:97"
        deactivateAt="csma.network.NetworkTest:103">
        <PrimField varName="isCollisionCheckEventProcessed" valToSet="false"/>
    </Fault>

<!-- Test Case 5: testRhoGreaterThanOrEqualToOneException -->
    <Fault class="csma.network.Network" lineNo="751" numOfInvoc="1"
        activateAt="csma.network.NetworkTest:145"
        deactivateAt="csma.network.NetworkTest:159">
        <PrimLocal varName="arrivalRate" valToSet="100"/>
    </Fault>

</FaultConfig>

```

D-2 ARS FAULT CONFIGURATION FILE (CASE STUDY II, PART 1)

```
<!--
  ARS case-study #2, part 1
    developed by Chris Acantilado and Neil Acantilado
-->

<FaultConfig>

  <Fault class="ars.database.DatabaseManager" lineNo="142" enable="false">
    <ObjField varName="arsDBStatement" setToNull="true"/>
  </Fault>

  <Fault class="ars.database.DatabaseManager" lineNo="160" enable="true" >
    <ObjField varName="arsDBStatement" setToNull="true"/>
  </Fault>

</FaultConfig>
```

D-3 ARS FAULT CONFIGURATION FILE (CASE STUDY II, PART 2)

```
<!--
  Gretel/ARS case-study 2, part 3 (GUI freezing)
    developed by Chris Acantilado and Neil Acantilado
-->

<FaultConfig>

  <Fault class="ars.database.DatabaseManager" lineNo="628" enable="true">
    <ObjLocal varName="result" setToNull="true"/>
  </Fault>

</FaultConfig>
```

D-4 GRETEL/ARS FAULT CONFIGURATION FILE (CASE STUDY III)

```
<!--  
    ARS case-study 3  
    developed by Chris Acantilado and Neil Acantilado  
-->  
  
<FaultConfig>  
  
    <Fault class="ars.database.DatabaseManager" lineNo="658">  
        <Exception prob="1.0" arg="0"/>  
    </Fault>  
  
</FaultConfig>
```


THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E – SIMPLE SOURCE CODE

This appendix contains the working source code of SIMPLE build that was used in the case studies. As evident, it is a work in progress.

E-1 BUILD.XML

```
<project name="simple" default="compile">

  <!-- Define some properties for external libraries used by this project -->
  <target name="setProperties">
    <property name="JAVA_HOME" value="c:\j2sdk1.4.0"/>
    <property name="TOOLS_PATH" value="c:\j2sdk1.4.0\lib"/>
    <property name="JPDA_PATH" value="c:\thesis\dev\JPDA\"/>
    <property name="ASPECTJ_PATH" value="c:\java\aspectj1.0\lib"/>
    <property name="JAVASSIST_PATH" value="c:\java\javassist2.0"/>
    <property name="JUNIT_PATH" value="c:\java\junit3.7"/>
    <property name="SRC_DIR" value="\"/>
    <property name="XERCES_PATH" value="c:\java\xerces-2_0_1"/>
    <property name="JTREK_PATH" value="c:\java\jtrek"/>
  </target>

  <taskdef name="ajc" classname="org.aspectj.tools.ant.taskdefs.Ajc">
    <classpath>
      <pathelement location="${ASPECTJ_PATH}/aspectjtools.jar"/>
      <pathelement location="${ASPECTJ_PATH}/aspectj-ant.jar"/>
      <pathelement location="${JAVA_HOME}/lib/tools.jar"/>
    </classpath>
  </taskdef>

  <target name="compileJPDA" depends="setProperties">
    <javac srcdir="${JPDA_PATH}" source="1.4" excludes="Hello.java">
      <classpath>
        <pathelement location="\"/>
        <pathelement location="${JPDA_PATH}" />
        <pathelement location="${TOOLS_PATH}/tools.jar"/>
        <pathelement location="${ASPECTJ_PATH}/aspectjrt.jar"/>
        <pathelement location="${JAVA_HOME}/lib/tools.jar"/>
        <pathelement location="${JAVASSIST_PATH}/javassist.jar"/>
        <pathelement location="${JUNIT_PATH}/junit.jar"/>
        <pathelement location="${XERCES_PATH}/xercesImpl.jar"/>
        <pathelement location="${XERCES_PATH}/xercesSamples.jar"/>
        <pathelement location="${XERCES_PATH}/xmlParserAPIs.jar"/>
        <pathelement location="${JTREK_PATH}" />
      </classpath>
    </javac>
  </target>

  <target name="compile" depends="compileJPDA">
    <ajc srcdir="${SRC_DIR}" source="1.4">
      <classpath>
        <pathelement location="${SRC_DIR}" />
        <pathelement location="${JPDA_PATH}" />
        <pathelement location="${TOOLS_PATH}/tools.jar"/>
        <pathelement location="${ASPECTJ_PATH}/aspectjrt.jar"/>
        <pathelement location="${JAVA_HOME}/lib/tools.jar"/>
        <pathelement location="${JAVASSIST_PATH}/javassist.jar"/>
        <pathelement location="${JUNIT_PATH}/junit.jar"/>
        <pathelement location="${XERCES_PATH}/xercesImpl.jar"/>
        <pathelement location="${XERCES_PATH}/xercesSamples.jar"/>
        <pathelement location="${XERCES_PATH}/xmlParserAPIs.jar"/>
        <pathelement location="${JTREK_PATH}" />
      </classpath>
    </ajc>
  </target>

  <target name="cleanall">
    <delete>
```

```
        <fileset dir="." includes="**/*.class"/>
      </delete>
    </target>

    <target name="cleanbuild" depends="cleanall, compile" />
  </project>
```

E-2 DOM_UTIL.JAVA

```
package simple.util;
import java.io.*;
import org.apache.xerces.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

// Source code adapted from the book Program Generators with XML and Java
// by J. Craig Cleaveland

/**
 * The DOM_Util class provides high-level API to parse an XML document.
 *
 * @author    various (available as open-source)
 * @created   May 1, 2002
 */
public class DOM_Util
{
    /**
     * Gets the attr attribute of the DOM_Util class
     *
     * @param n          Description of the Parameter
     * @param attrName    Description of the Parameter
     * @param defaultVal  Description of the Parameter
     * @return           The attr value
     */
    public static String getAttr(Node n, String attrName, String defaultVal)
    {
        if (n instanceof Document)
        {
            n = ((Document) n).getDocumentElement();
        }
        String v = null;
        if (n instanceof Element)
        {
            v = ((Element) n).getAttribute(attrName);
        }
        if (v == null || v.equals(""))
        {
            return defaultVal;
        }
        return v;
    }

    /**
     * Gets the intAttr attribute of the DOM_Util class
     *
     * @param n          Description of the Parameter
     * @param tagName     Description of the Parameter
     * @param defaultValue Description of the Parameter
     * @return           The intAttr value
     */
    public static int getIntAttr(Node n, String tagName, int defaultValue)
    {
        String s = getAttr(n, tagName, "");
        return parseInt(s, defaultValue);
    }

    /**
     * Description of the Method
     *
     * @param n          Description of the Parameter
     */
}
```

```

    *@param tagName      Description of the Parameter
    *@param defaultValue Description of the Parameter
    *@return             Description of the Return Value
    */
    public static String get(Node n, String tagName, String defaultValue)
    {
        if (n instanceof Document)
        {
            n = ((Document) n).getDocumentElement();
        }
        if (n instanceof Element)
        {
            NodeList nodes = ((Element) n).getElementsByTagName(tagName);
            if (nodes.getLength() == 0)
            {
                return defaultValue;
            }
            else
            {
                return getContent(nodes.item(0));
            }
        }
        else
        {
            return defaultValue;
        }
    }
}

/**
 * Gets the int attribute of the DOM_Util class
 *
 * @param n      Description of the Parameter
 * @param tagName Description of the Parameter
 * @param defaultValue Description of the Parameter
 * @return       The int value
 */
public static int getInt(Node n, String tagName, int defaultValue)
{
    String s = get(n, tagName, "");
    return parseInt(s, defaultValue);
}

/**
 * Gets the content attribute of the DOM_Util class
 *
 * @param n Description of the Parameter
 * @return  The content value
 */
public static String getContent(Node n)
{
    StringBuffer buf = new StringBuffer();
    getContent1(n, buf);
    return buf.toString();
}

/**
 * Description of the Method
 *
 * @param filename      Description of the Parameter
 * @return              Description of the Return Value
 * @exception Exception Description of the Exception
 */
public static Document readDocument(String filename) throws Exception

```

```

{
    DOMParser parser = new DOMParser();
    parser.parse(new InputSource(new FileInputStream(filename)));
    return parser.getDocument();
}

/**
 * Description of the Method
 *
 * @param s      Description of the Parameter
 * @param defaultValue Description of the Parameter
 * @return      Description of the Return Value
 */
private static int parseInt(String s, int defaultValue)
{
    int returnValue;
    try
    {
        returnValue = Integer.parseInt(s);
    }
    catch (NumberFormatException exc)
    {
        returnValue = defaultValue;
    }
    return returnValue;
}

/**
 * Gets the content1 attribute of the DOM_Util class
 *
 * @param n      Description of the Parameter
 * @param buf    Description of the Parameter
 */
private static void getContent1(Node n, StringBuffer buf)
{
    for (Node c = n.getFirstChild(); c != null; c = c.getNextSibling())
    {
        if (c instanceof Element || c instanceof EntityReference)
        {
            getContent1(c, buf);
        }
        else if (c instanceof Text)
        {
            buf.append(c.getNodeValue());
        }
    }
}
}

```

E-3 EVENTTHREAD.JAVA

```
/*
 *  @(#) EventThread.java 1.3 01/12/03
 *
 *  Copyright 2002 Sun Microsystems, Inc. All rights reserved.
 *  SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */
/*
 *  Copyright (c) 1997-2001 by Sun Microsystems, Inc. All Rights Reserved.
 *
 *  Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 *  modify and redistribute this software in source and binary code form,
 *  provided that i) this copyright notice and license appear on all copies of
 *  the software; and ii) Licensee does not utilize the software in a manner
 *  which is disparaging to Sun.
 *
 *  This software is provided "AS IS," without a warranty of any kind. ALL
 *  EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING
 *  ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 *  OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT
 *  BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING,
 *  MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL
 *  SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR
 *  DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 *  HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF
 *  THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF
 *  THE POSSIBILITY OF SUCH DAMAGES.
 *
 *  This software is not designed or intended for use in on-line control of
 *  aircraft, air traffic, aircraft navigation or aircraft communications; or
 *  in the design, construction, operation or maintenance of any nuclear
 *  facility. Licensee represents and warrants that it will not use or
 *  redistribute the Software for such purposes.
 */
package simple;

import simple.fault.*;
import simple.util.*;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import java.util.*;

/**
 * This class processes incoming JDI events and displays them
 *
 * @author Neil Acantilado
 * @author Chris Acantilado
 * @created April 7, 2002
 * @version
 * @(#) EventThread.java 1.3 01/12/03 00:15:38
 */
public class EventThread extends Thread
{
    // Classes that will be excluded from the class prepare process ... Should
    // make the tester specify this ...
    private final static String[] excludes = {"java.*", "javax.*", "com.sun.*",
        "sun.*", "junit.*", "dec.*"};
```



```

// Running VM
private final VirtualMachine vm;

// Connected to VM
private boolean connected = true;

// VMDeath occurred
private boolean vmDied = true;

private EventRequestManager eventRequestManager = null;

// Holds the startTime
private long startTime = -1;

// Fault Manager that is used to manage and execute faults accordingly
private FaultManager faultManager = null;

/**
 * Constructor for the EventThread object
 *
 * @param vm Running JVM
 */
EventThread(VirtualMachine vm)
{
    super("Debugger Event-Handler");

    this.vm = vm;
    this.eventRequestManager = vm.eventRequestManager();
    this.faultManager = new FaultManager();
}

/**
 * Adds a fault to the Fault Manager
 *
 * @param fault Fault to be added to the event-thread
 */
public void addFault(Fault fault)
{
    faultManager.add(fault);
}

/**
 * Deletes a fault from the Fault Manager
 *
 * @param fault Fault to be removed from the event-thread
 */
public void removeFault(Fault fault)
{
    faultManager.remove(fault);
}

/**
 * Sets the startTime attribute of the EventThread object
 */
public void setStartTime()
{
    // I understand that this can be expensive ...
    startTime = System.currentTimeMillis();
}

/**
 * Run the event handling thread. As long as we are connected, get event

```

```

    * sets off the queue and dispatch the events within them.
    */
public void run()
{
    addClassPrepareRequest();

    setStartTime();

    EventQueue queue = vm.eventQueue();

    while (connected)
    {
        try
        {
            EventSet eventSet = queue.remove();
            EventIterator it = eventSet.eventIterator();
            while (it.hasNext())
            {
                handleEvent(it.nextEvent());
            }
            eventSet.resume();
        }
        catch (InterruptedException exc)
        {
            // Ignore
        }
        catch (VMDisconnectedException discExc)
        {
            handleDisconnectedException();
            break;
        }
    }
    System.exit(1);
}

/**
 * Adds a feature to the ClassPrepareRequest attribute of the EventThread
 * object
 */
private void addClassPrepareRequest()
{
    ClassPrepareRequest cpr =
        eventRequestManager.createClassPrepareRequest();
    for (int i = 0; i < excludes.length; i++)
    {
        cpr.addClassExclusionFilter(excludes[i]);
    }
    cpr.setSuspendPolicy(EventRequest.SUSPEND_ALL);
    cpr.enable();
}

/**
 * Sets the breakpointEvents attribute of the EventThread object
 *
 * @param location Indicates the location to apply the breakpoint request
 * @return Description of the Return Value
 */
private EventRequest addBreakpointRequest(Location location)
{
    BreakpointRequest bpr =
        eventRequestManager.createBreakpointRequest(location);
    bpr.setSuspendPolicy(EventRequest.SUSPEND_ALL);
    bpr.enable();
}

```

```

        return bpr;
    }

    /**
     * Sets the stepEventRequest attribute of the EventThread object
     *
     * @param thread Indicates the thread to apply the step request
     * @return Description of the Return Value
     */
    private EventRequest addStepRequest(ThreadReference thread)
    {
        StepRequest req = eventRequestManager.createStepRequest(thread,
            StepRequest.STEP_LINE, StepRequest.STEP_INT0);

        for (int i = 0; i < excludes.length; i++)
        {
            req.addClassExclusionFilter(excludes[i]);
        }

        // Hard-coded for now
        req.addClassExclusionFilter("org.aspectj.*");
        req.addClassExclusionFilter("simple.*");

        req.setSuspendPolicy(EventRequest.SUSPEND_ALL);
        req.enable();
        return req;
    }

    /**
     * Sets the methodEntryEventRequest attribute of the EventThread object
     *
     * @param excludes The classes to ignore for method entry requests
     * @param classPattern The class patterns to consider
     * @return Description of the Return Value
     */
    private EventRequest addMethodEntryRequest(String[] excludes,
        String classPattern)
    {
        MethodEntryRequest menr =
            eventRequestManager.createMethodEntryRequest();
        for (int i = 0; i < excludes.length; i++)
        {
            menr.addClassExclusionFilter(excludes[i]);
        }
        menr.addClassFilter(classPattern);
        menr.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        menr.enable();
        return menr;
    }

    /**
     * Sets the methodExitEventRequest attribute of the EventThread object
     *
     * @param excludes The classes to ignore for method exit requests
     * @return Description of the Return Value
     */
    private EventRequest addMethodExitRequest(String[] excludes)
    {
        MethodExitRequest mexr = eventRequestManager.createMethodExitRequest();
        for (int i = 0; i < excludes.length; ++i)
        {
            mexr.addClassExclusionFilter(excludes[i]);
        }
    }

```

```

        mexr.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        mexr.enable();
        return mexr;
    }

    /**
     * Adds a feature to the ModificationWatchpointRequest attribute of the
     * EventThread object
     *
     * @param field The field to track
     * @return      Description of the Return Value
     */
    private EventRequest addModificationWatchpointRequest(Field field)
    {
        ModificationWatchpointRequest req =
            eventRequestManager.createModificationWatchpointRequest(field);
        req.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        req.enable();
        return req;
    }

    /**
     * Adds a feature to the AccessWatchpointRequest attribute of the
     * EventThread object
     *
     * @param field The field to track
     * @return      Description of the Return Value
     */
    private EventRequest addAccessWatchpointRequest(Field field)
    {
        AccessWatchpointRequest req =
            eventRequestManager.createAccessWatchpointRequest(field);
        req.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        req.enable();
        return req;
    }

    /**
     * Dispatch incoming events
     *
     * @param event The remote event from the target JVM
     */
    private void handleEvent(Event event)
    {
        if (event instanceof BreakpointEvent)
        {
            breakpointEvent((BreakpointEvent) event);
        }
        else if (event instanceof StepEvent)
        {
            stepEvent((StepEvent) event);
        }
        else if (event instanceof ExceptionEvent)
        {
            exceptionEvent((ExceptionEvent) event);
        }
        else if (event instanceof ModificationWatchpointEvent)
        {
            modificationWatchpointEvent((ModificationWatchpointEvent) event);
        }
        else if (event instanceof AccessWatchpointEvent)
        {
            accessWatchpointEvent((AccessWatchpointEvent) event);
        }
    }

```

```

    }
    else if (event instanceof MethodEntryEvent)
    {
        methodEntryEvent((MethodEntryEvent) event);
    }
    else if (event instanceof MethodExitEvent)
    {
        methodExitEvent((MethodExitEvent) event);
    }
    else if (event instanceof ThreadDeathEvent)
    {
        threadDeathEvent((ThreadDeathEvent) event);
    }
    else if (event instanceof ClassPrepareEvent)
    {
        classPrepareEvent((ClassPrepareEvent) event);
    }
    else if (event instanceof VMStartEvent)
    {
        vmStartEvent((VMStartEvent) event);
    }
    else if (event instanceof VMDeathEvent)
    {
        vmDeathEvent((VMDeathEvent) event);
    }
    else if (event instanceof VMDisconnectEvent)
    {
        vmDisconnectEvent((VMDisconnectEvent) event);
    }
    else
    {
        throw new Error("Unexpected event type");
    }
}

/**
 * Process breakpoint events
 *
 * @param event The breakpoint event in question
 */
private void breakpointEvent(BreakpointEvent event)
{
    String locationDescriptor = event.location().toString();
    ThreadReference thread = event.thread();
    faultManager.execute(locationDescriptor, thread, startTime);
}

/**
 * A VMDisconnectedException has happened while dealing with another
 * event. We need to flush the event queue, dealing only with exit events
 * (VMDeath, VMDisconnect) so that we terminate correctly.
 */
synchronized void handleDisconnectedException()
{
    EventQueue queue = vm.eventQueue();
    while (connected)
    {
        try
        {
            EventSet eventSet = queue.remove();
            EventIterator iter = eventSet.eventIterator();
            while (iter.hasNext())
            {

```

```

        Event event = iter.nextEvent();
        if (event instanceof VMDeathEvent)
        {
            vmDeathEvent((VMDeathEvent) event);
        }
        else if (event instanceof VMDisconnectEvent)
        {
            vmDisconnectEvent((VMDisconnectEvent) event);
        }
    }
    // Resume the VM
    eventSet.resume();
}
catch (InterruptedException exc)
{
    // ignore
}
}

/**
 * Processes VMStartEvents
 *
 * @param event The VMStartEvent in question
 */
private void vmStartEvent(VMStartEvent event) { }

/**
 * Processes methodEntryEvents
 *
 * @param event The MethodEntryEvent in question
 */
private void methodEntryEvent(MethodEntryEvent event) { }

/**
 * Processes methodExitEvents
 *
 * @param event The MethodExitEvent in question
 */
private void methodExitEvent(MethodExitEvent event) { }

/**
 * Processes StepEvents
 *
 * @param event The StepEvent in question
 */
private void stepEvent(StepEvent event)
{
    //System.out.println(event.location()); // NPA
}

/**
 * Processes ModificationWatchpointEvents
 *
 * @param event The ModificationWatchpointEvent in question
 */
private void modificationWatchpointEvent(ModificationWatchpointEvent event)
{ }

/**
 * Processes AccessWatchpointEvents
 *
 * @param event The AccessWatchpointEvent in question

```

```

    */
private void accessWatchpointEvent(AccessWatchpointEvent event) { }

/**
 * Processes ThreadDeathEvents
 *
 * @param event The ThreadDeathEvent in question
 */
void threadDeathEvent(ThreadDeathEvent event) { }

/**
 * A new class has been loaded. Set watchpoints on each of its fields
 *
 * @param event The ClassPrepareEvent in question
 */
private void classPrepareEvent(ClassPrepareEvent event)
{
    try
    {
        ReferenceType refType = event.referenceType();

        // Store a reference to the ReferenceType
        SimpleRepository.addClassType((ClassType) refType);
        //System.out.println(refType.name()); // NPA

        // Resolve preconfigured breakpoints ...
        List locations = refType.allLineLocations();
        for (Iterator iter = locations.iterator(); iter.hasNext(); )
        {
            Location location = (Location) iter.next();
            String descriptor = location.toString();

            if (faultManager.containsLocation(descriptor))
            {
                System.out.println(new StringBuffer()
                    .append("-- Breakpoint set at ")
                    .append(location).append(" --"));

                // Create the request ...
                EventRequest eventRequest = addBreakpointRequest(location);

                // Add the event request to the helper ... to be used for
                // other purposes ... Kinda ugly, though ...
                SimpleRepository.addEventRequest(descriptor, eventRequest);

                // Reset the state of the fault mapped to the descriptor...
                faultManager.reset(descriptor);

                // Determine if it needs to be initially disabled
                if (!faultManager.isEnabled(descriptor))
                {
                    eventRequest.disable();
                }
            }
        }

        if (refType.name().equals(SimpleTrek.SIMPLE_CLIENT_CLASS))
        {
            faultManager.setSimpleHelperClassType((ClassType) refType);
        }
    }
    catch (Exception e)
    {

```

```

        //e.printStackTrace(); // NPA
        //ignore
    }
}

/**
 * Processes ExceptionEvents
 *
 * @param event The ExceptionEvent in question
 */
private void exceptionEvent(ExceptionEvent event) { }

/**
 * Processes VMDeathEvents
 *
 * @param event The VMDeathEvent in question
 */
public void vmDeathEvent(VMDeathEvent event)
{
    vmDied = true;
}

/**
 * Processes VMDisconnectEvents
 *
 * @param event The VMDisconnectEvent in question
 */
public void vmDisconnectEvent(VMDisconnectEvent event)
{
    connected = false;
}
}

```


E-4 FAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import java.util.*;

/**
 * Abstract Fault class that encompasses logic for a general SIMPLE fault
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created      July 30, 2002
 */
public abstract class Fault
{
    /**
     * Constant indicating an infinite value
     */
    public final static int INDEFINITE = -1;

    /**
     * Constant indicating a random value
     */
    public final static String RANDOM_VALUE = "RANDOM_VALUE";

    /**
     * Used to generate default faultNames
     */
    private static int counter = 0;

    /**
     * The designated name of the fault
     */
    protected String faultName = null;

    /**
     * The Class to apply the fault in.
     */
    protected String className = null;

    /**
     * The exact line number of the fault occurrence
     */
    protected int lineNo = -1;

    /**
     * A combination of the className and lineNo ... Used as a key to the
     * fault hashMap objects in the Fault Manager ...
     */
    protected String descriptor = null;

    /**
     * Used to indicate the how many time the fault will occur
     */
    protected int numOfInvocations = INDEFINITE;

    /**
```

```

    * Used as a current counter for calculating numOfInvocations
    */
protected int currNumOfInvocations = INDEFINITE;

/**
 * Defines the probability of the fault occurrence
 */
protected double probability = 1.0;

/**
 * Defines the start time of the fault.
 */
protected long startTime = INDEFINITE;

/**
 * Defines the end time of the fault.
 */
protected long endTime = INDEFINITE;

/**
 * Used for enabling/disabling the fault
 */
protected boolean isEnabled = true;

/**
 * Description of the Field
 */
protected boolean isEnabledSetting = true;

/**
 * Description of the Field
 */
protected boolean isLocationActivated = false;

/**
 * Constructor for the Fault object
 *
 * @param className Class name to apply fault to
 * @param lineNo    Source line to apply fault to
 * @param faultName Description of the Parameter
 */
public Fault(String faultName, String className, int lineNo)
{
    this.faultName = faultName;
    this.className = className;
    this.lineNo = lineNo;
    this.descriptor = className + ":" + lineNo;

    if (faultName == null || faultName.length() == 0)
    {
        this.faultName = "fault " + counter;
        counter++;
    }
}

/**
 * Constructor for the Fault object
 *
 * @param className Description of the Parameter
 * @param lineNo    Description of the Parameter
 */
public Fault(String className, int lineNo)
{

```

```

        this.faultName = "fault " + counter;
        counter++;
        this.className = className;
        this.lineNo = lineNo;
        this.descriptor = className + ":" + lineNo;
    }

    /**
     * Gets the faultName attribute of the Fault object
     *
     * @return    The faultName value
     */
    public String getFaultName()
    {
        return faultName;
    }

    /**
     * Gets the className attribute of the Fault object
     *
     * @return    The className value
     */
    public String getClassName()
    {
        return className;
    }

    /**
     * Gets the lineNo attribute of the Fault object
     *
     * @return    The lineNo value
     */
    public int getLineNo()
    {
        return lineNo;
    }

    /**
     * Gets the locDescriptor attribute of the Fault object
     *
     * @return    The locDescriptor value
     */
    public String getDescriptor()
    {
        return descriptor;
    }

    /**
     * Sets the numOfInvocations attribute of the Fault object
     *
     * @param    numOfInvocations    The new numOfInvocations value
     */
    public void setNumOfInvocations(int numOfInvocations)
    {
        this.numOfInvocations = numOfInvocations;
        this.currNumOfInvocations = numOfInvocations;
    }

    /**
     * Gets the numOfInvocations attribute of the Fault object
     *
     * @return    The numOfInvocations value

```

```

    */
    public int getNumOfInvocations()
    {
        return numOfInvocations;
    }

    /**
     * Gets the currNumOfInvocations attribute of the Fault object
     *
     * @return The numOfInvocations value
     */
    public int getCurrNumOfInvocations()
    {
        return this.currNumOfInvocations;
    }

    /**
     * Sets the probability attribute of the Fault object
     *
     * @param probability The new probability value
     */
    public void setProbability(double probability)
    {
        this.probability = probability;
    }

    /**
     * Gets the probability attribute of the Fault object
     *
     * @return The probability value
     */
    public double getProbability()
    {
        return probability;
    }

    /**
     * Sets the injectionStartTime attribute of the Fault object
     *
     * @param startTime The new startTime value
     */
    public void setStartTime(long startTime)
    {
        this.startTime = startTime;
    }

    /**
     * Gets the injectionStartTime attribute of the Fault object
     *
     * @return The injectionStartTime value
     */
    public long getStartTime()
    {
        return startTime;
    }

    /**
     * Sets the injectionEndTime attribute of the Fault object
     *
     * @param endTime The new endTime value
     */
    public void setEndTime(long endTime)

```

```

{
    this.endTime = endTime;
}

/**
 * Gets the injectionEndTime attribute of the Fault object
 *
 * @return    The injectionEndTime value
 */
public long getEndTime()
{
    return endTime;
}

/**
 * Determines whether the fault can be injected given the specified
 * boundary times ... Dependent upon probability and currNumOfInvocations
 *
 * @param    currentTime    Description of the Parameter
 * @return    Description of the Return Value
 */
public boolean timeToInject(long currentTime)
{
    // If the fault has been disabled, then return immediately ... This
    // doesn't mean that it should be expired ...
    if (!isEnabled)
    {
        return false;
    }

    // If there aren't any more invocations, expire the fault ...
    if (currNumOfInvocations == 0)
    {
        isEnabled = false;
        return false;
    }

    if (Math.random() > probability)
    {
        return false;
    }

    // The below will cause a true
    if (startTime < 0 || currentTime >= startTime)
    {
        if (endTime < 0 || currentTime < endTime)
        {
            // Update the current tally of invocations ...
            if (currNumOfInvocations > 0)
            {
                currNumOfInvocations--;
            }
            return true;
        }

        // If we are at this point, then the fault should be expired ...
        isEnabled = false;
    }

    return false;
}

/**

```

```

    * Gets the expired attribute of the Fault object
    *
    *@return    The expired value
    */
public boolean isExpired(long currentTime)
{
    // Check to see if the fault has expired ...
    if (currNumOfInvocations == 0 || endTime > currentTime)
    {
        isEnabled = false;
        return true;
    }
    return false;
}

/**
 * Sets the enabledSetting attribute of the Fault object
 *
 *@param  isEnabledSetting  The new enabledSetting value
 */
public void setEnabledSetting(boolean isEnabledSetting)
{
    this.isEnabledSetting = isEnabledSetting;
}

/**
 * Sets the enable attribute of the Fault object
 *
 *@param  isEnabled  The new enabled value
 */
public void setEnabled(boolean isEnabled)
{
    this.isEnabled = isEnabled;
}

/**
 * Gets the enabled attribute of the Fault object
 *
 *@return    The enabled value
 */
public boolean isEnabled()
{
    return isEnabled;
}

/**
 * Sets the locationActivated attribute of the Fault object
 *
 *@param  isLocationActivated  The new locationActivated value
 */
public void setLocationActivated(boolean isLocationActivated)
{
    this.isLocationActivated = isLocationActivated;
}

/**
 * Gets the locationActivated attribute of the Fault object
 *
 *@return    The locationActivated value
 */
public boolean isLocationActivated()
{
    return isLocationActivated;
}

```

```

    }

    /**
     * The method that resets the state of the fault. Fault Subclasses must
     * provide actual implementation.
     */
    public void reset()
    {
        currNumOfInvocations = numOfInvocations;
        isEnabled = (isEnabledSetting && !isLocationActivated);
    }

    /**
     * The method that gets executed. Fault Subclasses must provide actual
     * implementation.
     *
     * @param thread      Thread that breakpoint was invoked in
     * @param vm          Virtual Machine that SUT is running under
     * @param frame        Stack frame that breakpoint was invoked in
     * @param objectReference ObjectReference passed in by event-thread
     * @param currentTime  Current time passed in by event-thread
     */
    public abstract void execute(ThreadReference thread, VirtualMachine vm,
        StackFrame frame, ObjectReference objectReference, long currentTime);
}

```

E-5 FAULTMANAGER.JAVA

```
package simple;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;
import java.util.*;

import simple.fault.*;
import simple.util.*;

import java.util.List;

/**
 * The Fault Manager is responsible for the management and execution of
 * user-defined faults.
 *
 * @author      Neil Acantialdo
 * @author      Chris Acantialdo
 * @created     May 1, 2002
 */
public class FaultManager
{
    // Values are ArrayList objects
    private HashMap cfgFaultMap = new HashMap();

    // Values are Fault array objects ... Used for fast iteration ...
    private HashMap excFaultMap = new HashMap();

    // Just a temp variable
    private Fault[] temp = new Fault[0];

    // Holds the ObjectReference to the client-side SFI helper class
    private ObjectReference simpleHelperObjectReference = null;

    // Holds the ClassType to the client-side SFI helper class
    private ClassType simpleHelperClassType = null;

    // Holds a Method reference to the client-side SFI helper class methods
    private Method simpleHelperConstructor = null;

    // Holds a Method reference to the client-side SFI helper class methods
    private Method simpleHelperMethod = null;

    // Temp arraylist to represent no arguments when a method is invoked
    private final static ArrayList noArgs = new ArrayList();

    /**
     * Determines whether time is handled by the Test Harness or by the
     * client. Performance is better when time is handled by the client, but
     * this means invoking the pre-instrumentation tool on relevant SUT
     * classes ... Trade-off here ...
     */
    public static boolean isTimeInstrumentedOnClient = false;

    /**
     * Description of the Field
     */
    public long overhead = 0; // NPA -- 062602
}
```



```

/**
 * Sets the simpleHelperObjectReference attribute (and others) of the
 * FaultManager object.
 *
 * @param simpleHelperClassType The new simpleHelperClassType value
 */
public void setSimpleHelperClassType(ClassType simpleHelperClassType)
{
    // Access the client-side SIMPLE helper and keep appropriate references
    // to it so that SIMPLE can communicate with it during Fault-Injection
    // testing ...
    this.simpleHelperClassType = simpleHelperClassType;

    // Reinitializes reflective components to be used later
    initSimpleHelperComponents();
    // NPA -- 070902

    // Initialize to null, so we can find its reference each time the
    // fault-injection tests are re-executed
    simpleHelperObjectReference = null;
}

/**
 * Reinitializes reflective components
 */
private void initSimpleHelperComponents()
{
    simpleHelperConstructor = null;
    simpleHelperMethod = null;

    // Check if exists -- NPA 072502
    if (simpleHelperClassType == null)
    {
        return;
    }

    List methods = simpleHelperClassType.allMethods();
    int size = methods.size();

    for (int i = 0; i < size; i++)
    {
        // Don't bother going on if we already have what we need. This
        // SUT could be pretty lengthy ...
        if (simpleHelperConstructor != null && simpleHelperMethod != null)
        {
            break;
        }

        Method method = (Method) methods.get(i);
        if (method.name().equals("getClientCurrentTime"))
        {
            simpleHelperMethod = method;
        }
        else if (method.toString().indexOf(SimpleTrek.SIMPLE_CLIENT_CLASS
            + "<init>") != -1)
        {
            simpleHelperConstructor = method;
        }
    }
}

/**
 * Basically answers the question: Does fault bucket exist for this

```

```

    * descriptor?
    *
    * @param descriptor Location descriptor of the fault
    * @return           True if fault bucket exists. False, otherwise.
    */
    public boolean containsLocation(String descriptor)
    {
        return cfgFaultMap.containsKey(descriptor);
    }

    /**
     * Adds a fault to the appropriate fault bucket.
     *
     * @param fault The fault to add
     */
    public void add(Fault fault)
    {
        String descriptor = fault.getDescriptor();
        ArrayList faultBucket = (ArrayList) cfgFaultMap.get(descriptor);

        if (faultBucket == null)
        {
            // Create a fault bucket if it doesn't exist yet
            faultBucket = new ArrayList();
            cfgFaultMap.put(descriptor, faultBucket);
        }

        faultBucket.add(fault);

        // Add the fault to the fault bucket ...
        // Simultaneously update the excFaultMap ...
        excFaultMap.remove(descriptor);
        excFaultMap.put(descriptor, faultBucket.toArray(temp));
    }

    /**
     * Removes the fault
     *
     * @param fault The fault to be removed
     */
    public void remove(Fault fault)
    {
        String descriptor = fault.getDescriptor();
        ArrayList faultBucket = (ArrayList) cfgFaultMap.get(descriptor);
        if (faultBucket != null)
        {
            // Remove the fault to the bucket ...
            faultBucket.remove(fault);
            // Simultaneously update the excFaultMap ...
            excFaultMap.remove(descriptor);
            excFaultMap.put(descriptor, faultBucket.toArray(temp));
        }
        else
        {
            excFaultMap.remove(descriptor);
        }
    }

    /**
     * Executes the faults associated with the location accordingly. Need to
     * try to minimize the overhead here ...
     */

```

```

    *@param descriptor The location descriptor describing the breakpoint.
    *@param thread      The thread that the breakpoint was invoked.
    *@param startTime   The start time
    */
    public void execute(String descriptor, ThreadReference thread,
        long startTime)
    {
        Fault[] faults = (Fault[]) excFaultMap.get(descriptor);
        if (faults != null)
        {
            try
            {
                long currentTime = -1;

                // Figure out an approximate current time ... (expensive)
                // This depends on what type of time handling was selected by
                // the tester/developer
                if (isTimeInstrumentedOnClient)
                {
                    // Time is handled by the SUT
                    if (simpleHelperObjectReference == null)
                    {
                        simpleHelperObjectReference =
                            simpleHelperClassType.newInstance(thread,
                                simpleHelperConstructor, noArgs, 0);
                    }

                    LongValue currentTimeValue = (LongValue)
                        simpleHelperObjectReference.invokeMethod(thread,
                            simpleHelperMethod, noArgs, 0);
                    currentTime = currentTimeValue.longValue();
                }
                else
                {
                    // Time is handled by the SIMPLE
                    currentTime = System.currentTimeMillis() - startTime;
                    currentTime -= overhead; // NPA -- 062602
                }

                long overheadStart = System.currentTimeMillis();
                // NPA -- 062602

                // Get vm, frame, and objectReference from the thread ...
                VirtualMachine vm = thread.virtualMachine();

                StackFrame frame = thread.frame(0);
                ObjectReference objectReference = thread.frame(0).thisObject();

                // Boolean used for determining whether a breakpoint can be
                // disabled or not ...(NPA -- 07/27/02)
                boolean disableFault = true;

                // Execute the faults for this descriptor ...
                for (int i = 0; i < faults.length; i++)
                {
                    faults[i].execute(thread, vm, frame, objectReference,
                        currentTime);

                    // Check if the breakpoint needs to be active due
                    // any outstanding faults that are still enabled ...
                    // That is, we don't want to deactivate the breakpoint if
                    // faults are still being injected ...(NPA -- 07/27/02)
                    if (!faults[i].isExpired(currentTime))

```

```

        {
            disableFault = false;
        }
    }

    // If it's the case that ALL faults at this breakpoint are no
    // longer active, then remove the breakpoint ...
    // (NPA -- 07/27/02)
    if (disableFault)
    {
        SimpleRepository.disableEventRequest(descriptor);
    }

    long overheadEnd = System.currentTimeMillis(); // NPA -- 062602
    overhead += (overheadEnd - overheadStart); // NPA -- 062602
}
catch (Exception e)
{
    // Ignore for now ...
}
}

}

/**
 * Resets all the faults to their uninitialized states
 *
 * @param descriptor Location descriptor of the breakpoint
 */
public void reset(String descriptor)
{
    overhead = 0;
    // NPA -- 062602

    simpleHelperObjectReference = null;
    //simpleHelperConstructor = null; // NPA -- 070902
    //simpleHelperMethod = null; // NPA -- 070902
    initSimpleHelperComponents();
    // NPA -- 070902

    Fault[] faults = (Fault[]) excFaultMap.get(descriptor);
    if (faults != null)
    {
        // Execute the faults for this descriptor ...
        for (int i = 0; i < faults.length; i++)
        {
            faults[i].reset();
        }
    }
}

/**
 * Gets the enabled attribute of the FaultManager object
 *
 * @param descriptor Description of the Parameter
 * @return The enabled value
 */
public boolean isEnabled(String descriptor)
{
    boolean enable = false;

    Fault[] faults = (Fault[]) excFaultMap.get(descriptor);
    if (faults != null)
    {

```

```

        // Execute the faults for this descriptor ...
        for (int i = 0; i < faults.length; i++)
        {
            // As long as one fault is active out of a set of many, then
            // the breakpoint needs to be enabled ...
            if (faults[i].isEnabled())
            {
                enable = true;
            }
        }
    }
    return enable;
}

/**
 * Gets the faults attribute of the FaultManager object
 *
 * @param descriptor Description of the Parameter
 * @return The faults value
 */
//public Fault[] getFaults(String descriptor)
//{
//    Fault[] faults = (Fault[]) excFaultMap.get(descriptor);
//    if (faults == null)
//    {
//        faults = temp;
//    }
//    return faults;
//}
}

```

E-6 FAULTPARSER.JAVA

```
package simple;

import com.sun.jdi.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.util.*;
import java.lang.*;

import simple.fault.*;
import simple.util.*;

/**
 * The FaultParser class will parse an XML file for desired faults configured
 * by the tester/developer. Uses a DOM utility to extract this information
 * from the fault config file. This class is a bit long and definitely is in
 * need of improvement.
 *
 * @author Neil Acantilado
 * @author Chris Acantilado
 * @created April 27, 2002
 */
public class FaultParser
{
    // Used to store the parsed faults and will be passed to the Fault-Manager
    // when finished.
    private Collection faultList = new ArrayList();

    // Used to instrument byte-code of particular classes of the SUT
    private SimpleTrek simpleTrek = new SimpleTrek();

    private boolean instrumentOnly = false;

    private String filename = "faults.xml";

    /**
     * Constructor for the FaultParser class
     *
     * @param filename Filename of Fault XML file
     * @exception Exception Description of the Exception
     */
    public FaultParser(String filename) throws Exception
    {
        this.filename = filename;
    }

    /**
     * Gets the faults attribute of the FaultParser object
     *
     * @return The faults value
     */
    public Fault[] getFaults()
    {
        return (Fault[]) faultList.toArray(new Fault[0]);
    }

    /**
     * Starts parsing of XML-configured faults
     *
     * @param d Document to parse ...
     * @return Description of the Return Value
     */
}
```

```

public boolean convertDocument(Document d)
{
    Element faultsNode = (Element) d.getDocumentElement();

    //
    // Process the "InstrumentOption" node elements ...
    //
    NodeList instrumentOptionNodeList =
        faultsNode.getElementsByTagName("InstrumentOption");
    for (int i = 0; i < instrumentOptionNodeList.getLength(); i++)
    {
        Node instrumentOptionNode = instrumentOptionNodeList.item(i);

        String booleanStr = DOM_Util.getAttr(instrumentOptionNode,
            "isTimeInstrumentedOnClient", "false");

        instrumentOnly = DOM_Util.getAttr(instrumentOptionNode,
            "isInstrumentOnly", "false").equals("true");

        FaultManager.isTimeInstrumentedOnClient =
            Boolean.valueOf(booleanStr).booleanValue();
    }

    //
    // Process the "Fault" node elements ...
    //
    NodeList faultNodeList = faultsNode.getElementsByTagName("Fault");
    for (int i = 0; i < faultNodeList.getLength(); i++)
    {
        Node faultNode = faultNodeList.item(i);

        // Will ignore fault-entry entirely ... That is, the fault will NOT
        // be added to the FaultManager ...
        String ignore = DOM_Util.getAttr(faultNode, "ignore", "false");
        if (ignore.equals("true"))
        {
            continue;
        }

        String faultName = DOM_Util.getAttr(faultNode, "name", "");
        String className = DOM_Util.getAttr(faultNode, "class", "");
        String lineNoStr = DOM_Util.getAttr(faultNode, "lineNo", "-1");
        String numOfInvocStr = DOM_Util.getAttr(faultNode, "numOfInvoc",
            "-1");
        String probStr = DOM_Util.getAttr(faultNode, "prob", "1.0");
        String startTimeStr = DOM_Util.getAttr(faultNode, "startTime",
            "-1");
        String endTimeStr = DOM_Util.getAttr(faultNode, "endTime", "-1");
        String enableSetting = DOM_Util.getAttr(faultNode, "enable",
            "true");
        String activateAt = DOM_Util.getAttr(faultNode, "activateAt", "");
        String deactivateAt = DOM_Util.getAttr(faultNode, "deactivateAt",
            "");

        int lineNo = Integer.parseInt(lineNoStr);
        int numOfInvoc = Integer.parseInt(numOfInvocStr);
        double prob = Double.parseDouble(probStr);
        long startTime = Long.parseLong(startTimeStr) * 1000;
        long endTime = Long.parseLong(endTimeStr) * 1000;
        boolean enable = enableSetting.equals("true");

        Fault fault = null;
        if (faultNode.hasChildNodes())

```

```

{
    // Process the individual faults ...
    NodeList subFaultNodeList = faultNode.getChildNodes();
    for (int j = 0; j < subFaultNodeList.getLength(); j++)
    {
        Node subFaultNode = subFaultNodeList.item(j);
        String faultType = subFaultNode.getNodeName();

        // Process PrimLocalFaultNode ...
        if (faultType.indexOf("Prim") != -1)
        {
            fault = processPrimitiveFault(subFaultNode,
                faultName, className, lineNo, numOfInvoc, prob,
                startTime, endTime, enable);
            add(fault, activateAt, deactivateAt);
        }

        // Process ObjLocalFaultNode ...
        else if (faultType.indexOf("Obj") != -1)
        {
            fault = processObjectFault(subFaultNode,
                faultName, className, lineNo, numOfInvoc, prob,
                startTime, endTime, enable);
            add(fault, activateAt, deactivateAt);
        }

        // Exception Fault ...
        else if (faultType.equals("Exception"))
        {
            processSpecializedFault("Exception", subFaultNode,
                className, lineNo, numOfInvoc, prob, startTime,
                endTime, enable);
        }

        // Memory Exhaust Fault ...
        else if (faultType.equals("Memory"))
        {
            processSpecializedFault("Memory", subFaultNode,
                className, lineNo, numOfInvoc, prob, startTime,
                endTime, enable);
        }

        // Processor Exhaust Fault ...
        else if (faultType.equals("Processor"))
        {
            processSpecializedFault("Processor", subFaultNode,
                className, lineNo, numOfInvoc, prob, startTime,
                endTime, enable);
        }

        // Processor Exhaust Fault ...
        else if (faultType.equals("Delay"))
        {
            processSpecializedFault("Delay", subFaultNode,
                className, lineNo, numOfInvoc, prob, startTime,
                endTime, enable);
        }
    }
}

// Process "hacked" faults that are really time indicators ...
NodeList startTimeNodeList =

```



```

faultsNode.getElementsByTagName("StartTime");
for (int i = 0; i < startTimeNodeList.getLength(); i++)
{
    Node startTimeNode = startTimeNodeList.item(i);

    // Will ignore fault-entry entirely ... That is, the fault will NOT
    // be added to the FaultManager ...
    String ignore = DOM_Util.getAttr(startTimeNode, "ignore", "false");
    if (ignore.equals("true"))
    {
        continue;
    }

    // Do not process the node if fault is indicated to be not enabled.
    String enable = DOM_Util.getAttr(startTimeNode, "enable", "true");

    // Process information essential to breakpoint configuration.
    String className = DOM_Util.getAttr(startTimeNode, "class", "");
    String lineNoStr = DOM_Util.getAttr(startTimeNode, "lineNo", "-1");
    int lineNo = Integer.parseInt(lineNoStr);

    // Add the "hacked" node into the fault list ...
    // Need to fix this ...
    if (FaultManager.isTimeInstrumentedOnClient)
    {
        simpleTrek.instrument(className, lineNo,
            SimpleTrek.SIMPLE_TIME_METHOD);
    }
    else
    {
        StartTime startTime = new StartTime(className, lineNo);
        startTime.setProbability(1.0);
        startTime.setEnabled(enable.equals("true"));
        // NPA -- 072502
        faultList.add(startTime);
    }
}

if (!FaultManager.isTimeInstrumentedOnClient)
{
    UpdateTime updateTime = new UpdateTime();
    updateTime.setEnabled(true);
    // NPA -- 072502
    faultList.add(updateTime);
}

return instrumentOnly;
}

/**
 * Process local-level faults. These faults corrupt the primitive local
 * variables with a specified method of a class
 *
 * @param faultName Description of the Parameter
 * @param className Description of the Parameter
 * @param lineNo Description of the Parameter
 * @param numOfInvoc Description of the Parameter
 * @param prob Description of the Parameter
 * @param startTime Description of the Parameter
 * @param endTime Description of the Parameter
 * @param enable Description of the Parameter
 * @param faultNode Description of the Parameter
 * @return Description of the Return Value

```

```

    */
private Fault processPrimitiveFault(Node faultNode, String faultName,
    String className, int lineNo, int numOfInvoc, double prob,
    long startTime, long endTime, boolean enable)
{
    String faultType = faultNode.getNodeName();
    String varName = DOM_Util.getAttr(faultNode, "varName", "");
    String valToSet = DOM_Util.getAttr(faultNode, "valToSet",
        Fault.RANDOM_VALUE);

    PrimitiveFault fault = null;
    if (faultType.equals("PrimLocal"))
    {
        fault = new PrimitiveLocalFault(faultName, className, lineNo,
            varName);
    }
    else
    {
        fault = new PrimitiveFieldFault(faultName, className, lineNo,
            varName);
    }

    fault.setValueToSet(valToSet);
    fault.setNumOfInvocations(numOfInvoc);
    fault.setProbability(prob);
    fault.setStartTime(startTime);
    fault.setEndTime(endTime);
    fault.setEnabledSetting(enable);

    return fault;
}

/**
 * Process field-level object faults. These faults corrupt the field
 * object member variables within a class
 *
 * @param className Name fo class to apply the fault
 * @param lineNo Line number within class to apply the fault
 * @param faultName Description of the Parameter
 * @param numOfInvoc Description of the Parameter
 * @param prob Description of the Parameter
 * @param startTime Description of the Parameter
 * @param endTime Description of the Parameter
 * @param enable Description of the Parameter
 * @param faultNode Description of the Parameter
 * @return The final constructed fault
 */
private Fault processObjectFault(Node faultNode, String faultName,
    String className, int lineNo, int numOfInvoc, double prob,
    long startTime, long endTime, boolean enable)
{
    String faultType = faultNode.getNodeName();
    String varName = DOM_Util.getAttr(faultNode, "varName", "");
    String setToNull = DOM_Util.getAttr(faultNode, "setToNull", "false");

    ObjectFault objectFault = null;
    if (faultType.equals("ObjLocal"))
    {
        objectFault = new ObjectLocalFault(faultName, className, lineNo,
            varName);
    }
    else

```

```

    {
        objectFault = new ObjectFieldFault(faultName, className, lineNo,
            varName);
    }

    objectFault.setProbability(prob);
    objectFault.setNumOfInvocations(numOfInvoc);
    objectFault.setStartTime(startTime);
    objectFault.setEndTime(endTime);
    objectFault.setEnabledSetting(enable);
    objectFault.setToNull(setToNull.equals("true"));

    NodeList subFaultNodeList = faultNode.getChildNodes();
    for (int k = 0; k < subFaultNodeList.getLength(); k++)
    {
        Node subFaultNode = subFaultNodeList.item(k);
        String nodeName = subFaultNode.getNodeName();

        Fault fault = null;
        if (nodeName.equals("ObjField"))
        {
            fault = processObjectFault(subFaultNode, faultName, className,
                lineNo, numOfInvoc, prob, startTime, endTime, enable);
            objectFault.add(fault);
        }
        else if (nodeName.equals("PrimField"))
        {
            fault = processPrimitiveFault(subFaultNode, faultName,
                className, lineNo, numOfInvoc, prob, startTime, endTime,
                enable);
            objectFault.add(fault);
        }
    }

    return objectFault;
}

/**
 * This method will process the 'specialized' faults found within the
 * Fault config file. These faults are specialized due to the fact that
 * they invoke pre-instrumentation actions on the byte-code of compiled
 * SUT classes.
 *
 * @param identifier Identifies what type of fault: Exc, Mem, Dly, Prc
 * @param className The name of the class to instrument the fault
 * @param lineNo The line number to instrument the fault
 * @param faultNode Description of the Parameter
 * @param numOfInvoc Description of the Parameter
 * @param prob Description of the Parameter
 * @param startTime Description of the Parameter
 * @param endTime Description of the Parameter
 * @param enable Description of the Parameter
 */
private void processSpecializedFault(String identifier, Node faultNode,
    String className, int lineNo, int numOfInvoc, double prob, long
    startTime, long endTime, boolean enable)
{
    String argStr = DOM_Util.getAttr(faultNode, "arg", "-1");
    String whereToInjectStr = DOM_Util.getAttr(faultNode, "whereToInject",
        "before");
    int arg = Integer.parseInt(argStr);
    boolean isBefore = whereToInjectStr.equals("before");

```

```

// Instrument accordingly ...
if (identifier.equals("Exception"))
{
    simpleTrek.instrument(className, lineNo,
        SimpleTrek.SIMPLE_EXC_METHOD, startTime, endTime, prob,
        numOfInvoc, arg, isBefore);
}
else if (identifier.equals("Processor"))
{
    simpleTrek.instrument(className, lineNo,
        SimpleTrek.SIMPLE_PRC_METHOD, startTime, endTime, prob,
        numOfInvoc, arg, isBefore);
}
else if (identifier.equals("Memory"))
{
    simpleTrek.instrument(className, lineNo,
        SimpleTrek.SIMPLE_MEM_METHOD, startTime, endTime, prob,
        numOfInvoc, arg, isBefore);
}
else if (identifier.equals("Delay"))
{
    simpleTrek.instrument(className, lineNo,
        SimpleTrek.SIMPLE_DLY_METHOD, startTime, endTime, prob,
        numOfInvoc, arg, isBefore);
}
}

/**
 * Description of the Method
 *
 * @param fault      Description of the Parameter
 * @param activateAt Description of the Parameter
 * @param deactivateAt Description of the Parameter
 */
private void add(Fault fault, String activateAt, String deactivateAt)
{
    faultList.add(fault);

    if (activateAt.length() != 0)
    {
        fault.setLocationActivated(true);

        // Separate into class and line number
        int index = activateAt.indexOf(":");
        String name = activateAt.substring(0, index);
        String lineNoStr = activateAt.substring(index + 1,
            activateAt.length());
        int lineNo = Integer.parseInt(lineNoStr);

        LocationFaultTrigger trigger =
            new LocationFaultTrigger(name, lineNo, true);
        trigger.setFault(fault);
        trigger.setNumOfInvocations(fault.getNumOfInvocations());
        faultList.add(trigger);
    }

    if (deactivateAt.length() != 0)
    {
        // Separate into class and line number
        int index = deactivateAt.indexOf(":");
        String name = deactivateAt.substring(0, index);
        String lineNoStr = deactivateAt.substring(index + 1,

```

```

        deactivateAt.length());
int lineNo = Integer.parseInt(lineNoStr);

LocationFaultTrigger trigger =
    new LocationFaultTrigger(name, lineNo, false);
trigger.setFault(fault);
trigger.setNumOfInvocations(fault.getNumOfInvocations());
faultList.add(trigger);
    }
}

```

E-7 LOCATIONFAULTTRIGGER.JAVA

```
package simple.fault;

import simple.FaultManager;
import simple.util.*;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

/**
 * This class represents a breakpoint action where LocationFaultTrigger is set
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     July 25, 2002
 */
public class LocationFaultTrigger extends Fault
{
    private Fault fault = null;

    private boolean activate = true;

    /**
     * Constructor for the LocationFaultTrigger object
     *
     * @param faultName Description of the Parameter
     * @param className Description of the Parameter
     * @param lineNo     Description of the Parameter
     * @param activate   Description of the Parameter
     */
    public LocationFaultTrigger(String faultName, String className, int lineNo,
                               boolean activate)
    {
        super(faultName, className, lineNo);
        this.activate = activate;
    }

    /**
     * Constructor for the LocationFaultTrigger object
     *
     * @param className Description of the Parameter
     * @param lineNo     Description of the Parameter
     * @param activate   Description of the Parameter
     */
    public LocationFaultTrigger(String className, int lineNo, boolean activate)
    {
        super(className, lineNo);
        this.activate = activate;
    }

    /**
     * Constructor for the setFaultDescriptors object
     *
     * @param fault The new fault value
     */
    public void setFault(Fault fault)
    {
        this.fault = fault;
    }
}
```

```

/**
 * Gets the faultToTrigger attribute of the LocationFaultTrigger object
 *
 * @return    The faultToTrigger value
 */
public Fault getFault()
{
    return fault;
}

/**
 * Constructor for the execute object
 *
 * @param  vm          N/A
 * @param  frame        N/A
 * @param  objectReference  N/A
 * @param  currentTime  N/A
 * @param  thread       N/A
 */
public void execute(ThreadReference thread, VirtualMachine vm,
    StackFrame frame, ObjectReference objectReference, long currentTime)
{
    if (timeToInject(currentTime))
    {
        fault.setEnabled(activate);
        if (activate)
        {
            SimpleRepository.enableEventRequest(fault.getDescriptor());
        }
    }
}

/**
 * Resets faults back to original state
 */
public void reset()
{
    super.reset();
}
}

```

E-8 OBJECTFAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;
import java.util.*;

/**
 * The ObjectFault class contains common attributes and methods for the
 * ObjectFieldFault and ObjectVarFault classes
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     April 27, 2002
 */
public abstract class ObjectFault extends Fault
{
    /**
     * The name of the object attribute to corrupt
     */
    protected String variableName = null;

    /**
     * Will contain the primitive attribute faults of the object
     */
    protected Collection subFaults = new ArrayList();

    /**
     * Arrays will be used for expediency.
     */
    protected Fault[] subFaultArray = new Fault[0];

    /**
     * Determines whether the object itself is to be set to null
     */
    protected boolean setToNull = false;

    /**
     * Constructor for the ObjectFault object
     *
     * @param faultName    Description of the Parameter
     * @param className    Description of the Parameter
     * @param lineNo       Description of the Parameter
     * @param variableName Description of the Parameter
     */
    public ObjectFault(String faultName, String className, int lineNo,
        String variableName)
    {
        super(faultName, className, lineNo);
        this.variableName = variableName;
    }

    /**
     * Constructor for the ObjectFault object
     *
     * @param className    Name of class that fault is to be applied to
     * @param lineNo       Line number of class that fault is to be applied to
     * @param variableName Name of variable
     */
}
```



```

    */
    public ObjectFault(String className, int lineNo, String variableName)
    {
        super(className, lineNo);
        this.variableName = variableName;
    }

    /**
     * Sets the toNull attribute of the ObjectFault object
     *
     * @param setToNull The new toNull value
     */
    public void setToNull(boolean setToNull)
    {
        this.setToNull = setToNull;
    }

    /**
     * Gets the setToNull attribute of the ObjectFault object
     *
     * @return The setToNull value
     */
    public boolean isSetToNull()
    {
        return setToNull;
    }

    /**
     * Description of the Method
     *
     * @param fault Description of the Parameter
     */
    public void add(Fault fault)
    {
        subFaults.add(fault);
        // Add the fault
        // we now have to reset the array
        subFaultArray = (Fault[]) subFaults.toArray(new Fault[0]);
    }

    /**
     * Remove the fault
     *
     * @param fault The fault to remove
     */
    public void remove(Fault fault)
    {
        subFaults.remove(fault);
        // Remove the fault
        // we now have to reset the array
        subFaultArray = (Fault[]) subFaults.toArray(new Fault[0]);
    }

    /**
     * Resets fault state ...
     */
    public void reset()
    {
        // Reset all sub-faults
        for (int i = 0; i < subFaultArray.length; i++)
        {

```

```
        subFaultArray[i].reset();
    }
    super.reset();
}
```

E-9 OBJECTFIELDFFAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.event.*;
import com.sun.jdi.request.*;
import com.sun.tools.example.debug.bdi.*;
import com.sun.tools.example.debug.expr.*;

import java.util.*;

import simple.util.*;

/**
 * The ObjectFieldFault class represents faults that deal with field
 * attributes
 * that are not primitive types. That is, they are object types.
 *
 * @author Neil Acantilado
 * @author Chris Acantilado
 * @created April 27, 2002
 */
public class ObjectFieldFault extends ObjectFault
{
    // Cached to improve performance
    private TypeComponent field = null;

    // It could be the case that the frame is static where an ObjectReference
    // is not available. Thus, we need to keep a reference to the ClassType
    // around ... We get the class type from the Fault class where it keeps
    // an internal collection around for that purpose ... yeah, I know ...
    // This was added on after the fact ...
    private ClassType classType = null;

    /**
     * Constructor for the ObjectFieldFault object
     *
     * @param faultName Description of the Parameter
     * @param className Description of the Parameter
     * @param lineNo Description of the Parameter
     * @param variableName Description of the Parameter
     */
    public ObjectFieldFault(String faultName, String className, int lineNo,
        String variableName)
    {
        super(faultName, className, lineNo, variableName);
    }

    /**
     * Constructor for the ObjectFieldFault object
     *
     * @param className Name of class to apply fault to
     * @param lineNo Source line number of class to apply fault to
     * @param variableName Name of object variable to apply fault to
     */
    public ObjectFieldFault(String className, int lineNo, String variableName)
    {
        super(className, lineNo, variableName);
    }

    /**
     * Called by Fault-Manager to invoke faults

```

```

*
* @param vm          Virtual Machine provided by the JPDA
* @param objRef       ObjectReference
* @param frame        Frame that breakpoint was invoked
* @param thread       Description of the Parameter
* @param currentTime  The current time that time-stamped
*/
public void execute(ThreadReference thread, VirtualMachine vm,
    StackFrame frame, ObjectReference objRef, long currentTime)
{
    if (!timeToInject(currentTime))
    {
        // It's not time to inject yet ...
        return;
    }

    if (objRef == null)
    {
        // If we are here then the frame is a static frame ...
        if (classType == null)
        {
            classType = SimpleRepository.getClassType(className);
        }
        executeInStaticFrame(thread, vm, frame, classType, currentTime);
    }
    else
    {
        // If we are here then the frame is a non-static frame ...
        executeInNonStaticFrame(thread, vm, frame, objRef, currentTime);
    }
}

/**
 * Executes the fault as mandated by the Fault Manager (Non-Static Frame
 * Version)
 */
* @param vm          Virtual Machine provided by the JPDA
* @param objRef       ObjectReference
* @param frame        Frame that breakpoint was invoked
* @param thread       Description of the Parameter
* @param currentTime  The current time that time-stamped
*/
public void executeInNonStaticFrame(ThreadReference thread,
    VirtualMachine vm, StackFrame frame, ObjectReference objRef,
    long currentTime)
{
    // Gotta be careful when making changes in this method ...
    try
    {
        if (field == null)
        {
            // Get the Object attribute in question ...
            ReferenceType refTyp = objRef.referenceType();
            field = (Field) refTyp.fieldByName(variableName);
            if (field == null)
            {
                return;
            }
        }

        // At this point, we have the Object attribute that we
        // are searching for ...
    }
}

```

```

        // Set the Object attribute to NULL if configured to do so ...
        if (setToNull)
        {
            objRef.setValue((Field) field, null);
            return;
        }

        // Now corrupt the attributes of the object as defined by the
        // original fault definition ... We need to get its current
        // instance, first ...
        ObjectReference objectRef =
            (ObjectReference) objRef.getValue((Field) field);

        for (int i = 0; i < subFaultArray.length; i++)
        {
            subFaultArray[i].execute(thread, vm, frame, objectRef,
                currentTime);
        }
    }
    catch (Exception e)
    {
        // Ignore for now ...
    }
}

/**
 * Executes the fault as mandated by the Fault Manager (Static Frame
 * Version)
 *
 * @param vm          Virtual Machine provided by the JPDA
 * @param frame       Frame that breakpoint was invoked
 * @param thread      Description of the Parameter
 * @param currentTime The current time that time-stamped
 * @param classType   Description of the Parameter
 */
public void executeInStaticFrame(ThreadReference thread, VirtualMachine vm,
    StackFrame frame, ClassType classType, long currentTime)
{
    try
    {
        if (field == null)
        {
            field = (Field) classType.fieldByName(variableName);
            if (field == null)
            {
                return;
            }
        }

        // At this point, we have the Object attribute that we
        // are searching for ...
        if (setToNull)
        {
            // Here is where we set the object to null ...
            classType.setValue((Field) field, null);
            return;
        }

        // Now corrupt the attributes of the object as defined by the
        // original fault definition ... We need to get its current
        // instance first ...
        ObjectReference objectRef =
            (ObjectReference) classType.getValue((Field) field);

```

```

        for (int i = 0; i < subFaultArray.length; i++)
        {
            subFaultArray[i].execute(thread, vm, frame, objectRef,
                                     currentTime);
        }
    }
    catch (Exception e)
    {
        // Ignore for now ...
    }
}

/**
 * Reinitializes cached variables
 */
public void reset()
{
    field = null;
    classType = null;
    super.reset();
}
}

```

E-10 OBJECTLOCALFAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.event.*;
import com.sun.jdi.request.*;
import com.sun.tools.example.debug.bdi.*;
import com.sun.tools.example.debug.expr.*;

import java.util.*;

/**
 * The ObjectLocalFault classes represents faults that are applied to local
 * variables that are not primitives. That is, they are object instances.
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     April 27, 2002
 */
public class ObjectLocalFault extends ObjectFault
{
    // Cached to improve performance ...
    private LocalVariable localVar = null;

    /**
     * Constructor for the ObjectLocalFault object
     *
     * @param faultName      Description of the Parameter
     * @param className      Description of the Parameter
     * @param lineNo         Description of the Parameter
     * @param variableName   Description of the Parameter
     */
    public ObjectLocalFault(String faultName, String className, int lineNo,
        String variableName)
    {
        super(faultName, className, lineNo, variableName);
    }

    /**
     * Constructor for the ObjectLocalFault object
     *
     * @param className      Name of class to apply fault to
     * @param lineNo         Source line number of class to apply fault to
     * @param variableName   Name of object variable to apply fault to
     */
    public ObjectLocalFault(String className, int lineNo, String variableName)
    {
        super(className, lineNo, variableName);
    }

    /**
     * Executes the fault as mandated by the Fault Manager
     *
     * @param vm             Virtual Machine provided by the JPDA
     * @param objRef         ObjectReference
     * @param frame          Frame that breakpoint was invoked
     * @param thread         Description of the Parameter
     * @param currentTime    The current time that time-stamped
     */
    public void execute(ThreadReference thread, VirtualMachine vm,
        StackFrame frame, ObjectReference objRef, long currentTime)
```

```

{
    if (!timeToInject(currentTime))
    {
        // It's not time to inject yet ...
        return;
    }

    // Gotta be careful when making changes in this method ...
    try
    {
        if (localVar == null)
        {
            localVar = frame.visibleVariableByName(variableName);
            if (localVar == null)
            {
                return;
            }
        }

        // Set the Object attribute to NULL if configured to do so ...
        if (setToNull)
        {
            frame.setValue(localVar, null);
            return;
        }

        // Now corrupt the attributes of the object as defined by the
        // original fault definition ... We need to get its current
        // instance first ...
        ObjectReference objectRef =
            (ObjectReference) frame.getValue(localVar);
        for (int i = 0; i < subFaultArray.length; i++)
        {
            subFaultArray[i].execute(thread, vm, frame, objectRef,
                currentTime);
        }
    }
    catch (Exception e)
    {
        // Ignore for now ...
    }
}

/**
 * Resets cached variables
 */
public void reset()
{
    localVar = null;
    super.reset();
}
}

```


E-11 PRIMITIVEFAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import java.util.*;

import simple.util.Util;

/**
 * Abstract class that encompasses common attributes and methods for the
 * PrimitiveFieldFault and PrimitiveVariableFault classes.
 */
@author      Neil Acantilado
@author      Chris Acantilado
@created     April 27, 2002
*/
public abstract class PrimitiveFault extends Fault
{
    /**
     * Name of the variable (either field or local)
     */
    protected String variableName = null;

    /**
     * Textual description of the value to be set.
     */
    protected String valToSet = null;

    /**
     * Actual value to be set ... (Serves as a cache for better performance)
     */
    protected Value valueToSet = null;

    /**
     * Constructor for the PrimitiveFault object
     */
    @param faultName    Description of the Parameter
    @param className    Description of the Parameter
    @param lineNo       Description of the Parameter
    @param variableName Description of the Parameter
    */
    public PrimitiveFault(String faultName, String className, int lineNo,
        String variableName)
    {
        super(faultName, className, lineNo);
        this.variableName = variableName;
    }

    /**
     * Constructor for the PrimitiveFault object
     */
    @param className    Name of class to apply fault to
    @param lineNo       Source line of class to apply fault to
    @param variableName Name of the variable in question
    */
    public PrimitiveFault(String className, int lineNo,
        String variableName)
```

```

{
    super(className, lineNo);
    this.variableName = variableName;
}

/**
 * Sets the valueToSet attribute of the PrimitiveFault object
 *
 * @param valToSet The new valueToSet value
 */
public void setValueToSet(String valToSet)
{
    // Again, this is the textual description
    this.valToSet = valToSet;
}

/**
 * Gets the valueToSet attribute of the PrimitiveFault object
 *
 * @param vm          The Virtual Machine supplied by the JPDA for the
 *                    breakpoint
 * @param typeName    Description of the Parameter
 * @return            The valueToSet value
 */
public Value getValueToSet(String typeName, VirtualMachine vm)
{
    // Calculate a random value and return it.
    if (valToSet.equals(RANDOM_VALUE))
    {
        return Util.createRandomValue(typeName, vm);
    }

    // If a cached value doesn't exist, create one ...
    if (valueToSet == null)
    {
        return Util.createValue(typeName, valToSet, vm);
    }

    // Return the value
    return valueToSet;
}

/**
 * Resets all relevant values when tests are restarted
 */
public void reset()
{
    super.reset();
}
}

```

E-12 PRIMITIVEFIELDFAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import java.util.*;

import simple.util.*;

/**
 * The PrimitiveFieldFault class is responsible for accessing and corrupting
 * class member variables.
 */
@author      Neil Acantilado
@author      Chris Acantilado
@created     April 27, 2002
*/
public class PrimitiveFieldFault extends PrimitiveFault
{
    // Cached variables ... Attempts to improve performance.
    // Will be reset each time tests are reloaded
    private TypeComponent field = null;
    private String typeName = null;

    // It could be the case that the frame is static where an ObjectReference
    // is not available. Thus, we need to keep a reference to the ClassType
    // around ... We get the class type from the Fault class where it keeps
    // an internal collection around for that purpose ... yeah, I know ...
    // This was added on after the fact ...
    private ClassType classType = null;

    /**
     * Constructor for the PrimitiveFieldFault object
     */
    @param faultName    Description of the Parameter
    @param className    Description of the Parameter
    @param lineNo       Description of the Parameter
    @param variableName Description of the Parameter
    */
    public PrimitiveFieldFault(String faultName, String className, int lineNo,
        String variableName)
    {
        super(faultName, className, lineNo, variableName);
    }

    /**
     * Constructor for the PrimitiveFieldFault object
     */
    @param className    Classname to apply fault
    @param lineNo       Line number to apply fault at
    @param variableName Name of member variable
    */
    public PrimitiveFieldFault(String className, int lineNo, String
        variableName)
    {
        super(className, lineNo, variableName);
    }
}
```

```

/**
 * Upon each encountered breakpoint, this method will be executed when
 * invoked by the FaultManager
 *
 * @param vm          Virtual Machine where breakpoint occurred
 * @param frame        Stack frame where breakpoint occurred
 * @param objectReference ObjectReference
 * @param currentTime  The current time
 * @param thread       Thread where breakpoint occurred
 */
public void execute(ThreadReference thread, VirtualMachine vm,
    StackFrame frame, ObjectReference objectReference, long currentTime)
{
    if (!timeToInject(currentTime))
    {
        // It's not time to inject yet ...
        return;
    }

    if (objectReference == null)
    {
        // If we are here then the frame is static ...
        if (classType == null)
        {
            classType = SimpleRepository.getClassType(className);
        }
        executeInStaticFrame(thread, vm, frame, classType, currentTime);
    }
    else
    {
        // If we are here then the frame is a non-static ...
        executeInNonStaticFrame(thread, vm, frame, objectReference,
            currentTime);
    }
}

/**
 * Upon each encountered breakpoint, this method will be executed when
 * invoked by the FaultManager. (Non-Static Frame Version)
 *
 * @param vm          Virtual Machine where breakpoint occurred
 * @param frame        Stack frame where breakpoint occurred
 * @param objectReference ObjectReference
 * @param currentTime  The current time
 * @param thread       Thread where breakpoint occurred
 */
public void executeInNonStaticFrame(ThreadReference thread,
    VirtualMachine vm, StackFrame frame, ObjectReference objectReference,
    long currentTime)
{
    try
    {
        if (field == null)
        {
            // Search for the field ...
            ReferenceType refTyp = objectReference.referenceType();
            field = (Field) refTyp.fieldByName(variableName);
            if (field == null)
            {
                return;
            }
        }
    }
}

```

```

        if (typeName == null)
        {
            typeName = ((Field) field).type().toString();
        }

        // Get the designated value for the field ...
        Value value = getValueToSet(typeName, vm);
        if (value == null)
        {
            return;
        }

        // Set the value and update numOfIterations counter
        objectReference.setValue((Field) field, value);
    }
    catch (Exception e)
    {
        // Something happened ... ignore for now
    }
}

/**
 * Upon each encountered breakpoint, this method will be executed when
 * invoked by the FaultManager. (Static Frame Version)
 */
@param vm          Virtual Machine where breakpoint occurred
@param frame       Stack frame where breakpoint occurred
@param currentTime The current time
@param thread      Thread where breakpoint occurred
@param classType   Description of the Parameter
*/
public void executeInStaticFrame(ThreadReference thread,
    VirtualMachine vm, StackFrame frame, ClassType classType,
    long currentTime)
{
    try
    {
        if (field == null)
        {
            // Search for the field ...
            field = (Field) classType.fieldByName(variableName);
            if (field == null)
            {
                return;
            }
        }

        if (typeName == null)
        {
            typeName = ((Field) field).type().toString();
        }

        // Get the designated value for the field ...
        Value value = getValueToSet(typeName, vm);
        if (value == null)
        {
            return;
        }

        // Set the value and update numOfIterations counter
        classType.setValue((Field) field, value);
    }
    catch (Exception e)

```

```

        {
            // Something happened ... ignore for now
        }
    }

    /**
     * Resets cached variables
     */
    public void reset()
    {
        field = null;
        typeName = null;
        classType = null;
        super.reset();
    }
}

```

E-13 PRIMITIVELOCALFAULT.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import java.util.*;

/**
 * The PrimitiveVariableFault class is responsible for accessing and
 * corrupting local variables within a class' method.
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     April 27, 2002
 */
public class PrimitiveLocalFault extends PrimitiveFault
{
    // Cached variables ... Attempts to improve performance.
    // Will be reset each time tests are reloaded
    private LocalVariable localVariable = null;
    private String typeName = null;

    public PrimitiveLocalFault(String faultName, String className, int lineNo,
        String variableName)
    {
        super(faultName, className, lineNo, variableName);
    }

    /**
     * Constructor for the FieldFault object
     *
     * @param  className  Class that fault is to be applied to
     * @param  lineNo     Source line of class to apply fault
     * @param  variableName  Name of local variable to corrupt
     */
    public PrimitiveLocalFault(String className, int lineNo, String
        variableName)
    {
        super(className, lineNo, variableName);
    }

    /**
     * Upon each encountered breakpoint, this method will be executed when
     * invoked by the FaultManager
     *
     * @param  vm          Virtual Machine where breakpoint occurred
     * @param  frame       Stack frame where breakpoint occurred
     * @param  objectReference  ObjectReference
     * @param  currentTime  The current time
     * @param  thread      Thread where breakpoint occurred
     */
    public void execute(ThreadReference thread, VirtualMachine vm,
        StackFrame frame, ObjectReference objectReference, long currentTime)
    {
        // Gotta be careful when making changes in this method ...
        if (!timeToInject(currentTime))
        {
            return;
        }
    }
}
```

```

    }

    // Process class local elements ...
    try
    {
        // Search for the local ...
        if (localVariable == null)
        {
            localVariable = frame.visibleVariableByName(variableName);
            if (localVariable == null)
            {
                return;
            }
        }

        if (typeName == null)
        {
            typeName = localVariable.type().toString();
        }

        // Get the designated value for the local ...
        Value value = getValueToSet(typeName, vm);
        if (value == null)
        {
            return;
        }

        // Set the value and update numOfIterations counter
        frame.setValue(localVariable, value);
    }
    catch (Exception e)
    {
        // Something happened ... ignore for now
    }
}

/**
 * Resets cached variables
 */
public void reset()
{
    localVariable = null;
    typeName = null;
    super.reset();
}
}

```


E-14 SIMPLEHARNESS.JAVA

```
/*
 *  @(#) SimpleHarness.java          1.3 01/12/03
 *
 *  Copyright 2002 Sun Microsystems, Inc. All rights reserved.
 *  SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */
/*
 *  Copyright (c) 1997-2001 by Sun Microsystems, Inc. All Rights Reserved.
 *
 *  Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 *  modify and redistribute this software in source and binary code form,
 *  provided that i) this copyright notice and license appear on all copies of
 *  the software; and ii) Licensee does not utilize the software in a manner
 *  which is disparaging to Sun.
 *
 *  This software is provided "AS IS," without a warranty of any kind. ALL
 *  EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING
 *  ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 *  OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT
 *  BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING,
 *  MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL
 *  SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR
 *  DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 *  HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF
 *  THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF
 *  THE POSSIBILITY OF SUCH DAMAGES.
 *
 *  This software is not designed or intended for use in on-line control of
 *  aircraft, air traffic, aircraft navigation or aircraft communications; or
 *  in the design, construction, operation or maintenance of any nuclear
 *  facility. Licensee represents and warrants that it will not use or
 *  redistribute the Software for such purposes.
 */

package simple;

import com.sun.jdi.VirtualMachine;
import com.sun.jdi.Bootstrap;
import com.sun.jdi.connect.*;

import java.util.Map;
import java.util.List;
import java.util.Iterator;

import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

import simple.fault.*;
import simple.util.*;

/**
 * The main program for SIMPLE
 */
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     April 20, 2002
 */
public class SimpleHarness
{
    // Default config file to use ... Should make the tester specify this for
```

```

// added flexibility ...
private final static String DEFAULT_CONFIG_FILE = "Faults.xml";

// Running remote VM
private final VirtualMachine vm;

// The standard err I/O stream
private Thread errThread = null;

// The standard out I/O stream
private Thread outThread = null;

// The event-thread of the SUT JVM
private EventThread eventThread = null;

/**
 * The main program for SIMPLE
 *
 * @param args The command line arguments
 */
public static void main(String[] args)
{
    new SimpleHarness(args);
}

/**
 * Parse the command line arguments. Launch target VM. Apply faults to
 * target JVM.
 *
 * @param args Description of the Parameter
 */
SimpleHarness(String[] args)
{
    if (args.length < 1)
    {
        System.err.println("args missing");
        System.exit(1);
    }

    String filename = args[0];
    if (filename.indexOf(".xml") == -1)
    {
        System.err.println("Invalid xml arg");
        System.exit(1);
    }

    String arguments = "";
    for (int i = 1; i < args.length; i++)
    {
        arguments += args[i];
        arguments += " ";
    }

    vm = launchTarget(arguments);
    vm.setDebugTraceMode(0);
    eventThread = new EventThread(vm);
    redirectOutput();

    configureFaultsToInject(filename);

    eventThread.start();
    vm.resume();
}

```

```

        // Shutdown begins when event thread terminates
        shutdown();
    }

    /**
     * Description of the Method
     */
    private void shutdown()
    {
        try
        {
            eventThread.join();
            errThread.join();
            outThread.join();
        }
        catch (InterruptedException exc)
        {
            // we don't interrupt
        }
    }

    /**
     * Reads, parses, and configures Faults from the Faults xml file
     *
     * @param filename Description of the Parameter
     * @return          Description of the Return Value
     */
    public void configureFaultsToInject(String filename)
    {
        try
        {
            FaultParser faultParser = new FaultParser(filename);
            if (faultParser.convertDocument(DOM_Util.readDocument(filename)))
            {
                System.out.println("Pre-instrument only.");
                vm.exit(1);
                shutdown();
                System.exit(1);
            }

            Fault[] faults = faultParser.getFaults();
            for (int i = 0; i < faults.length; i++)
            {
                eventThread.addFault(faults[i]);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Redirects any output set forth by the SUT
     */
    private void redirectOutput()
    {
        Process process = vm.process();

        // Copy target's output and error to our output and error.
        errThread = new StreamRedirectThread("error reader",
            process.getErrorStream(), System.err);
        outThread = new StreamRedirectThread("output reader",

```

```

        process.getInputStream(), System.err);
    errThread.start();
    outThread.start();
}

/**
 * Launch target VM. Forward target's output and error.
 *
 * @param mainArgs command-line arguments
 * @return Virtual Machine that SUT will be running under
 */
private VirtualMachine launchTarget(String mainArgs)
{
    LaunchingConnector connector = findLaunchingConnector();
    Map arguments = connectorArguments(connector, mainArgs);
    try
    {
        return connector.launch(arguments);
    }
    catch (IOException exc)
    {
        throw new Error("Unable to launch target VM: " + exc);
    }
    catch (IllegalConnectorArgumentsException exc)
    {
        throw new Error("Internal error: " + exc);
    }
    catch (VMStartException exc)
    {
        throw new Error("Target VM failed to initialize: "
            + exc.getMessage());
    }
}

/**
 * Find a com.sun.jdi.CommandLineLaunch connector
 *
 * @return LaunchingConnector - Not sure what this is
 */
private LaunchingConnector findLaunchingConnector()
{
    List connectors = Bootstrap.virtualMachineManager().allConnectors();
    Iterator iter = connectors.iterator();
    while (iter.hasNext())
    {
        Connector connector = (Connector) iter.next();
        if (connector.name().equals("com.sun.jdi.CommandLineLaunch"))
        {
            return (LaunchingConnector) connector;
        }
    }
    throw new Error("No launching connector");
}

/**
 * Return the launching connector's arguments.
 *
 * @param connector Connector used to connect with target JVM
 * @param mainArgs Command-line arguments
 * @return Map of possible arguments
 */
private Map connectorArguments(LaunchingConnector connector,
    String mainArgs)

```

```
{
    Map arguments = connector.defaultArguments();
    Connector.Argument mainArg = (Connector.Argument)
arguments.get("main");
    if (mainArg == null)
    {
        throw new Error("Bad launching connector");
    }
    mainArg.setValue(mainArgs);

    return arguments;
}
}
```

E-15 SIMPLEHELPER.JAVA

```
package simple.util;
import dec.trek.*;

import java.util.*;

/**
 * Helper class to be instrumented into the target SUT. Instrumentation into
 * the SUT class files is necessary to minimize the remote JVM access by the
 * JPDA API.
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     May 14, 2002
 */
public class SimpleHelper
{
    // Constant indicating a Memory Exhaustion Fault
    private final static int MEM = 0;

    // Constant indicating a Processor Exhaustion Fault
    private final static int PRC = 1;

    // Constant indicating a Exception Fault
    private final static int EXC = 2;

    // Constant indicating a Delay Fault
    private final static int DLY = 3;

    // Used for calculating probabilities
    private final static Random random = new Random();

    // Holds object references generated via MemoryExhaustFaults ...
    private static ArrayList objectReferences = new ArrayList();

    // Holds thread references generated via ProcessorExhaustFaults ...
    private static ArrayList threadReferences = new ArrayList();

    // Holds fault references generated via ProcessorExhaustFaults ...
    private static Hashtable faultRequests = new Hashtable();

    // Holds starting time
    private static long clientStartTime = 0;

    // Holds overhead value
    private static long overhead = 0; // NPA -- 062602

    // Clear arrays each time this class is loaded ...
    static
    {
        reset();
    }

    /**
     * Resets all supplementary lists
     */
    public static void reset()
    {
        overhead = 0; // NPA -- 062602
        objectReferences.clear();
        threadReferences.clear();
        faultRequests.clear();
    }
}
```

```

}

/**
 * Determines whether a specified fault will be injected. Several factors
 * play a role in this. For instance, the start/end times, the
 * probability, and the number of iterations will be deciding factors.
 *
 * @param type          Type of fault
 * @param name          Name class where fault is located
 * @param line          Line number where fault is located
 * @param startTime     Start time to begin fault
 * @param endTime       End time to end fault
 * @param probability    Probability to determine chance to invoke fault
 * @param numIterations Number of times to invoke fault
 * @return              Determines whether to inject the fault or not
 */
public static boolean timeToInject(int type, String name, int line,
    long startTime, long endTime, double probability, int numIterations)
{
    // Get fault request. Create one if none is found.
    String key = new StringBuffer().append(type).append(name).append(":").
        append(line).toString();
    FaultRequest faultRequest = (FaultRequest) faultRequests.get(key);
    if (faultRequest == null)
    {
        faultRequest = new FaultRequest(numIterations);
        faultRequests.put(key, faultRequest);
    }

    // Any one of these conditions will return a false
    if (faultRequest.numOfIterations == 0 ||
        Math.random() > probability)
    {
        return false;
    }

    long currentTime = -1;
    // Break must be set below ...
    if (currentTime == -1)
    {
        currentTime = getClientCurrentTime();
    }

    // Check if fault is to be invoked depending upon defined start and
    // end times
    if (startTime < 0 || currentTime >= startTime)
    {
        if (endTime < 0 || currentTime < endTime)
        {
            // Decrement numOfIterations for the fault
            faultRequest.numOfIterations--;
            return true;
        }
    }

    return false;
}

/**
 * Resets the client time. Parameters are dummy args for now. Included to
 * help automatic processing of faults. Will fix later.
 *

```

```

    *@param name      N/A
    *@param line      N/A
    *@param startTime N/A
    *@param endTime   N/A
    *@param prob      N/A
    *@param numIter   N/A
    *@param arg       N/A
    */
    public static void resetClientStartTime(String name, int line, double prob,
        int numIter, long startTime, long endTime, int arg)
    {
        clientStartTime = System.currentTimeMillis();
        overhead = 0; // NPA -- 062602
        //System.out.println("StartTime = " + clientStartTime); // NPA
    }

    /**
     * Gets the clientCurrentTime attribute of the SimpleHelper class
     *
     * @return The clientCurrentTime value
     */
    private static long getClientCurrentTime()
    {
        long clientCurrentTime = System.currentTimeMillis() - clientStartTime;
        //System.out.println("CurrentTime = " + clientCurrentTime); // NPA
        return (clientCurrentTime - overhead); // NPA -- 062602
    }

    /**
     * Responsible for managing and executing Memory-Exhaustion faults.
     *
     * @param name      Name of class that fault will be invoked in
     * @param line      Source line of class to invoke fault
     * @param prob      Probability of fault
     * @param numIter   Number of times fault is to be invoked
     * @param startTime Starting time of fault
     * @param endTime   End time of fault
     * @param arg       Generic argument
     */
    public static void exhaustMemoryFault(String name, int line, double prob,
        int numIter, long startTime, long endTime, int arg)
    {
        long overheadStart = System.currentTimeMillis(); // NPA -- 062602
        if (timeToInject(MEM, name, line, startTime, endTime, prob, numIter))
        {
            //System.out.println("Memory Fault Invoked: " + arg); // NPA
            for (int i = 0; i < arg; i++)
            {
                objectReferences.add(new Object());
            }
        }
        long overheadEnd = System.currentTimeMillis(); // NPA -- 062602
        overhead += (overheadEnd - overheadStart); // NPA -- 062602
    }

    /**
     * Responsible for managing and executing Processor-Exhaustion faults.
     *
     * @param name      Name of class that fault will be invoked in
     * @param line      Source line of class to invoke fault
     * @param prob      Probability of fault

```



```

    *@param numIter    Number of times fault is to be invoked
    *@param startTime  Starting time of fault
    *@param endTime    End time of fault
    *@param arg        Generic argument
    */
    public static void exhaustProcessorFault(String name, int line,
        double prob, int numIter, long startTime, long endTime, int arg)
    {
        long overheadStart = System.currentTimeMillis(); // NPA -- 062602
        if (timeToInject(PRC, name, line, startTime, endTime, prob, numIter))
        {
            //System.out.println("Processor Fault Invoked: " + arg);//NPA
            for (int i = 0; i < arg; i++)
            {
                Thread thread = new ProcessorThread();
                threadReferences.add(thread);
                thread.start();
            }
        }
        long overheadEnd = System.currentTimeMillis(); // NPA -- 062602
        overhead += (overheadEnd - overheadStart); // NPA -- 062602
    }

    /**
     * Responsible for managing and executing Exception-type faults. Need to
     * add more exception types, though.
     */
    *@param name        Name of class that fault will be invoked in
    *@param line        Source line of class to invoke fault
    *@param prob        Probability of fault
    *@param numIter    Number of times fault is to be invoked
    *@param startTime  Starting time of fault
    *@param endTime    End time of fault
    *@param arg        Generic argument
    */
    public static void throwException(String name, int line, double prob,
        int numIter, long startTime, long endTime, int arg)
    {
        long overheadStart = System.currentTimeMillis(); // NPA -- 062602
        if (timeToInject(EXC, name, line, startTime, endTime, prob, numIter))
        {
            //System.out.println("Exception Fault Invoked: " + arg);//NPA
            switch (arg)
            {
                case 0:
                    throw new NullPointerException
                        ("NullPointerException generated by SIMPLE");
                case 1:
                    throw new OutOfMemoryError
                        ("OutOfMemoryError generated by SIMPLE");
                case 2:
                    throw new IndexOutOfBoundsException
                        ("IndexOutOfBoundsException generated by SIMPLE");
                case 3:
                    throw new ClassCastException
                        ("ClassCastException generated by SIMPLE");
                case 4:
                    throw new RuntimeException
                        ("RuntimeException generated by SIMPLE");
            }
        }
        long overheadEnd = System.currentTimeMillis(); // NPA -- 062602
    }

```

```

        overhead += (overheadEnd - overheadStart); // NPA -- 062602
    }

    /**
     * Responsible for managing and executing Delay-type faults. Need to add
     * more exception types, though.
     *
     * @param name      Name of class that fault will be invoked in
     * @param line      Source line of class to invoke fault
     * @param prob       Probability of fault
     * @param numIter    Number of times fault is to be invoked
     * @param startTime  Starting time of fault
     * @param endTime    End time of fault
     * @param arg        Generic argument
     */
    public static void forceDelay(String name, int line, double prob,
        int numIter, long startTime, long endTime, int arg)
    {
        long overheadStart = System.currentTimeMillis(); // NPA -- 062602
        if (timeToInject(DLY, name, line, startTime, endTime, prob, numIter))
        {
            //System.out.println("Delay Fault Invoked: " + arg);//NPA
            try
            {
                Thread.sleep(arg);
            }
            catch (Exception e)
            {
                // Ignore for now
            }
        }
        long overheadEnd = System.currentTimeMillis(); // NPA -- 062602
        overhead += (overheadEnd - overheadStart); // NPA -- 062602
    }

    /**
     * The FaultRequest class handles fault attributes for the Test-Harness.
     * It is created to help manage client-side faults instrumented by the
     * SIMPLE test harness.
     *
     * @author      Neil Acantilado
     * @author      Chris Acantilado
     * @created     May 22, 2002
     */
    private static class FaultRequest
    {
        /**
         * Number of iterations fault is to be invoked. Will be decremented
         * only upon successful execution of fault
         */
        public int numOfIterations = 0;

        /**
         * Constructs a new <code>FaultRequest</code> instance.
         *
         * @param numOfIterations  Number of times to invoke fault
         */
        public FaultRequest(int numOfIterations)
        {
            this.numOfIterations = numOfIterations;
        }
    }

```

```

}

/**
 * Thread objects that Processor-Exhaustion Faults will instantiate to
 * simulate CPU work.
 *
 * @author Neil Acantilado
 * @author Chris Acantilado
 * @created May 16, 2002
 */
private static class ProcessorThread extends Thread
{
    // Some counter to keep the thread busy ...
    private long counter = 0;

    /**
     * Main processing method for the ProcessorThread object
     */
    public void run()
    {
        while (true)
        {
            counter++;
            // Do we want to sleep here to relieve the CPU from time to
            // time?
        }
    }
}
}

```

E-16 SIMPLEREPOSITORY.JAVA

```
package simple.util;

import java.util.*;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import simple.fault.*;

/**
 * A sort of helper class that stores various objects
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     July 30, 2002
 */
public class SimpleRepository
{
    // Used to store classTypes in the case they are needed
    private static HashMap classTypes = new HashMap();

    // Used to store eventRequests in the case they are needed
    private static Hashtable eventRequests = new Hashtable();

    private static HashMap faults = new HashMap();

    /**
     * Adds a referenceType to the ClassType Hashtable
     *
     * @param classType The feature to be added to the ClassType attribute
     */
    public static void addClassType(ClassType classType)
    {
        classTypes.put(classType.name(), classType);
    }

    /**
     * Gets a referenceType from the ReferenceType Hashtable
     *
     * @param name Description of the Parameter
     * @return      The referenceType value
     */
    public static ClassType getClassType(String name)
    {
        return (ClassType) classTypes.get(name);
    }

    /**
     * Description of the Method
     *
     * @param descriptor The descriptor of the fault
     * @param eventRequest The event-request
     */
    public static void addEventRequest(String descriptor,
        EventRequest eventRequest)
    {
        eventRequests.put(descriptor, eventRequest);
    }
}
```

```

/**
 * Enables pending faults upon demand ...
 *
 * @param descriptor Description of the Parameter
 */
public static void enableEventRequest(String descriptor)
{
    EventRequest eventRequest = (EventRequest)
    eventRequests.get(descriptor);
    if (eventRequest != null)
    {
        eventRequest.enable();
    }
}

/**
 * Description of the Method
 *
 * @param descriptor Description of the Parameter
 */
public static void disableEventRequest(String descriptor)
{
    EventRequest eventRequest = (EventRequest)
    eventRequests.get(descriptor);
    if (eventRequest != null)
    {
        eventRequest.disable();
    }
}

/**
 * Description of the Method
 *
 * @param fault Description of the Parameter
 */
public static void addFault(Fault fault)
{
    faults.put(fault.getFaultName(), fault);
}

/**
 * Description of the Method
 *
 * @param faultName Description of the Parameter
 * @return Description of the Return Value
 */
public static Fault getFault(String faultName)
{
    return (Fault) faults.get(faultName);
}
}

```

E-17 SIMPLETREK.JAVA

```
package simple.util;

import dec.trek.*;
import java.io.*;

/**
 * The SimpleTrek class uses Compaq's JTrek API to pre-instrument particular
 * faults into the SUT application byte-code. More specifically, The faults
 * are the following: Memory Exhaustion, Processor Exhaustion, Delays, and
 * Exception Throwing.
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     May 16, 2002
 */
public class SimpleTrek extends Trek
{
    /**
     * Name of client-side class that helps manages SIMPLE faults.
     */
    public final static String SIMPLE_CLIENT_CLASS =
        "simple.util.SimpleHelper";

    /**
     * Name of client-side method that handles memory exhaustion faults
     */
    public final static String SIMPLE_MEM_METHOD = SIMPLE_CLIENT_CLASS +
        ".exhaustMemoryFault";

    /**
     * Name of client-side method that handles processor exhaustion faults
     */
    public final static String SIMPLE_PRC_METHOD = SIMPLE_CLIENT_CLASS +
        ".exhaustProcessFault";

    /**
     * Name of client-side method that handles thrown exception faults
     */
    public final static String SIMPLE_EXC_METHOD = SIMPLE_CLIENT_CLASS +
        ".throwException";

    /**
     * Name of client-side method that handles forced delay faults
     */
    public final static String SIMPLE_DLY_METHOD = SIMPLE_CLIENT_CLASS +
        ".forceDelay";

    /**
     * Name of client-side method that handles start time indicator markers
     */
    public final static String SIMPLE_TIME_METHOD = SIMPLE_CLIENT_CLASS +
        ".resetClientStartTime";

    /**
     * Line number of the client-side utility to set breakpoints to
     */
    public final static int SIMPLE_SOURCE_LINE = 101;

    // Name of the class to apply the instrumented fault
    private String className = null;
}
```

```

// Line of the class to apply the instrumented fault
private int line = -1;

// Name of the method to be instrumented into the SUT
private String methodToInsert = null;

// The starting time to activate the fault
private long startTime = -1;

// The ending time to activate the fault
private long endTime = -1;

// The probability that the fault will occur during a run
private double probability = 1.0;

// The number of times that the fault will be applied
private int numOfIterations = -1;

// The generic argument used by the specified client-side method. It is
// used for various purposes.
private int arg = -1;

// Determines whether fault is to be applied before or after the statement
// specified at the line number
private boolean isBefore = true;

// Boolean flag to determine whether fault has been applied successfully or
// not
private boolean resolved = false;

// Debug flag to enable/disable debug statements
private boolean debug = false;

// Utility helper class that attempts to synchronize byte-code line numbers
// with source code line numbers.
private StatementHelper statementHelper = null;

/**
 * Sets the debug attribute of the SimpleTrek object. Used for enabling or
 * disabling debug statements.
 *
 * @param debug The new debug value
 */
protected void setDebug(boolean debug)
{
    this.debug = debug;
}

/**
 * Method that invokes the pre-instrumentation phase for the automatic
 * configuration of faults. Convenience method derived from the original
 * instrument method.
 *
 * @param className Classname to apply faults
 * @param line Line number of class to apply faults
 * @param methodToInsert The client-side method to instrument
 * @param startTime The start time of faults
 * @param endTime The end time of faults
 * @param probability The fault probability
 * @param numOfIterations The number of times the fault is to occur
 * @param arg Generic argument used by the specified method
 * @param isBefore Determines where the faults should be applied
 */

```

```

public void instrument(String className, int line, String methodToInsert,
    long startTime, long endTime, double probability,
    int numOfIterations, int arg, boolean isBefore)
{
    this.className = className;
    this.line = line;
    this.methodToInsert = methodToInsert;
    this.startTime = startTime;
    this.endTime = endTime;
    this.probability = probability;
    this.numOfIterations = numOfIterations;
    this.arg = arg;
    this.isBefore = isBefore;
    this.resolved = false;

    try
    {
        // Trekkie stuff ...
        statementHelper = new StatementHelper(className);
        getCmdLine(new String[]{className, "-ip", "simple;.",
            "-op", "simple"});
        doTrek();
        endTrek();
        // close the streams
        statementHelper.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }

    // Print out a warning if fault could not be instrumented for some
    // reason
    if (!resolved)
    {
        System.out.println("ERROR: Cannot instrument class " + className
            + " at line " + line + ".");
    }
}

/**
 * Method that invokes the pre-instrumentation phase for the automatic
 * configuration of faults.
 *
 * @param className      Classname to apply faults
 * @param line           Line number of class to apply faults
 * @param methodToInsert The client-side method to instrument
 * @param startTime      The start time of faults
 * @param endTime       The end time of faults
 * @param probability    The fault probability
 * @param numOfIterations The number of times the fault is to occur
 * @param arg            Generic argument used by the specified method
 */
public void instrument(String className, int line, String methodToInsert,
    long startTime, long endTime, double probability,
    int numOfIterations, int arg)
{
    instrument(className, line, methodToInsert, startTime, endTime,
        probability, numOfIterations, arg, true);
}

/**
 * Method that invokes the pre-instrumentation phase for the automatic

```



```

* configuration of faults. Convenience method derived from the original
* instrument method.
*
* @param className      Classname to apply faults
* @param line           Line number of class to apply faults
* @param methodToInsert The client-side method to instrument
*/
public void instrument(String className, int line, String methodToInsert)
{
    instrument(className, line, methodToInsert, -1, -1, -1, -1, -1);
}

/**
 * Description of the Method
 *
 * @param s Description of the Parameter
 */
public void atStartOf(Statement s)
{
    // Skip statement if fault has already been resolved
    if (resolved)
    {
        return;
    }

    // With the statementHelper, try to identify precisely the line number
    // of this statement. It is essential that this be correct. Else,
    // all is lost ...
    int sourceNumber = statementHelper.getLineNumberOfStatement(s);

    //System.out.println(sourceNumber + ":" + s); // NPA

    if (debug && sourceNumber > 0 &&
        sourceNumber < statementHelper.getTotalNumberOfLines())
    {
        System.out.println(sourceNumber + ":" + s + " (TYPE = " +
            s.getType() + ")");
    }

    // If this is the statement we want, proceed with the
    // pre-instrumentation process
    if (sourceNumber == line)
    {
        String name = s.getMethod().getClassFile().toQualifiedName();
        if (name != null)
        {
            Call call = null;

            if (isBefore)
            {
                call = Call.addBefore(methodToInsert, s);
            }
            else
            {
                call = Call.addAfter(methodToInsert, s);
            }

            // Hard-wire the relevant arguments for the pre-instrumentation
            // method
            call.passString(className);
            call.passInt(line);
            call.passDouble(probability);
        }
    }
}

```

```

        call.passInt(numOfIterations);
        call.passLong(startTime);
        call.passLong(endTime);
        call.passInt(arg);

        // Clean up
        call.done();

        // Indicate that fault has been resolved
        saveTrek();
        resolved = true;

        // Indicate success to the tester ...
        System.out.println("Inserted " + methodToInsert);
        System.out.println("\tat " + sourceNumber + ":" + s + "\n");
    }
}

/**
 * The main program for the SimpleTrek class
 *
 * @param argv The command line arguments
 */
public static void main(String[] argv)
{
    SimpleTrek simpleTrek = new SimpleTrek();
    simpleTrek.setDebug(true);
    simpleTrek.instrument("TestProgram", -20, null, -1, -1, -1, -1, -1);
}
}

```

E-18 STARTTIME.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

/**
 * This class represents a breakpoint action where StartTime is set
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     April 28, 2002
 */
public class StartTime extends Fault
{
    /**
     * Constructor for the StartTime object
     *
     * @param  className  Description of the Parameter
     * @param  lineNo     Description of the Parameter
     */
    public StartTime(String className, int lineNo)
    {
        super(className, lineNo);
    }

    /**
     * Constructor for the execute object
     *
     * @param  vm          N/A
     * @param  frame        N/A
     * @param  objectReference N/A
     * @param  currentTime  N/A
     * @param  thread       N/A
     */
    public void execute(ThreadReference thread, VirtualMachine vm,
        StackFrame frame, ObjectReference objectReference, long currentTime)
    { }
}
```

E-19 STATEMENTHELPER.JAVA

```
package simple.util;

import dec.trek.*;
import java.io.*;
import java.util.*;

/**
 * The StatementHelper class attempts to synchronize the line number
 * information embedded within the byte-code with the actual line numbers
 * within the source code. It is especially important that this class does
 * this correctly since the fault configuration relies heavily on the tester
 * specifying correct line numbers.
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     May 21, 2002
 */
public class StatementHelper
{
    // The line number currently being processed.
    private int currentLineNumber = 0;

    // The name of the file being processed.
    private String fileName = null;

    // The FileReader instance provides the IO stream associated to the file
    private FileReader fileReader = null;

    // The bufferedReader chains the fileReader for convenience.
    private BufferedReader bufferedReader = null;

    // The total number of lines within the file.
    private int totalNumOfLines = 0;

    // The text associated with the current line number that is being
    // processed.
    private String currentText = null;

    // The index within the text that is currently being processed.
    private int currentTextIndex = 0;

    /**
     * Constructor
     *
     * @param className The class under inspection
     */
    public StatementHelper(String className)
    {
        if (className.indexOf('$') != -1)
        {
            // We have an inner class here ...
            int endIndex = className.indexOf('$');
            className = className.substring(0, endIndex);
        }

        fileName = className.replace('.', '/').concat(".java");

        try
        {
            // First check if the class file has been preinstrumented earlier

```

```

        fileReader = new FileReader(fileName);
        bufferedReader = new BufferedReader(fileReader);
    }
    catch (Exception e2)
    {
        System.out.println(e2.getMessage());
    }

    // Figure out the total number of lines in this class
    totalNumOfLines = getTotalNumberOfLines(fileName);

    System.out.println("Processing " + fileName + "...");
}

/**
 * Gets the totalNumberOfLines attribute of the StatementHelper object
 *
 * @param fileName The name of the file that corresponds to the class
 * @return The totalNumberOfLines value
 */
public int getTotalNumberOfLines(String fileName)
{
    totalNumOfLines = 0;

    try
    {
        // Setup the file I/O
        FileReader fr = new FileReader(fileName);
        BufferedReader br = new BufferedReader(fr);

        // Count each line
        while (br.readLine() != null)
        {
            totalNumOfLines++;
        }

        // Close all streams
        br.close();
        fr.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }

    return totalNumOfLines;
}

/**
 * Gets the totalNumberOfLines attribute of the StatementHelper object
 *
 * @return The totalNumberOfLines value
 */
public int getTotalNumberOfLines()
{
    return totalNumOfLines;
}

/**
 * Gets the lineNumberOfStatement attribute of the StatementHelper object
 *
 * @param stmt The statement whose line number we are trying to figure out
 * @return The lineNumberOfStatement value

```

```

    */
    /*
public int getLineNumberOfStatement(Statement stmt)
{
    String match = null;

    // Determine what kind of statement we are processing by querying its
    // type attribute.
    int type = stmt.getType();
    switch (type)
    {
        case Trek.ST_BREAK:
            match = "break";
            break;
        case Trek.ST_CASE:
            // Case label could be 'default' or 'case'
            match = "default";
            String tempStr = stmt.toString();
            if (tempStr != null && tempStr.indexOf("case") != -1)
            {
                match = "case";
            }
            break;
        case Trek.ST_CONTINUE:
            match = "continue";
            break;
        case Trek.ST_DO:
            match = "do";
            break;
        case Trek.ST_FOR:
            match = "for";
            break;
        case Trek.ST_IF:
            match = "if";
            break;
        case Trek.ST_TRY:
            match = "try";
            break;
        case Trek.ST_WHILE:
            match = "while";
            break;
        case Trek.ST_CATCH:
            match = "catch";
            break;
        default:
            // If its none of the above, then we can go ahead and ask
            // JTrek to get the line number for us ...
            int linum = stmt.getAdjustedSourceNumber();
            if (linum > 0)
            {
                // Advance to the line
                advanceToLineNumber(linum);
            }
            return linum;
    }

    // Need to return the precise source code line number that matches our
    // statement.
    return getLineNumberOfStatement(match);
}
    */

    /**

```

```

* Gets the lineNumberOfStatement attribute of the StatementHelper object
*
@param stmt The statement whose line number we are trying to figure out
@return The lineNumberOfStatement value
*/
public int getLineNumberOfStatement(Statement stmt)
{
    // Get the statement's line number ...
    int linum = stmt.getAdjustedSourceNumber();
    if (linum > 0)
    {
        // Advance file pointer appropriately ...
        advanceToLineNumber(linum);
        return linum;
    }
    else if (linum == -1)
    {
        // If it's a statement that doesn't have an associated internal
        // method line number, then return immediately
        return -1;
    }

    String match = null;

    // Determine what kind of statement we are processing by querying its
    // type attribute.
    int type = stmt.getType();
    switch (type)
    {
        case Trek.ST_BREAK:
            match = "break";
            break;
        case Trek.ST_CASE:
            // Case label could be 'default' or 'case'
            match = "default";
            String tempStr = stmt.toString();
            if (tempStr != null && tempStr.indexOf("case") != -1)
            {
                match = "case";
            }
            break;
        case Trek.ST_CONTINUE:
            match = "continue";
            break;
        case Trek.ST_DO:
            match = "do";
            break;
        case Trek.ST_FOR:
            match = "for";
            break;
        case Trek.ST_IF:
            match = "if";
            break;
        case Trek.ST_TRY:
            match = "try";
            break;
        case Trek.ST_WHILE:
            match = "while";
            break;
        case Trek.ST_CATCH:
            match = "catch";
            break;
        default:

```

```

        return -1;
    }

    // Need to return the precise source code line number that matches our
    // statement.
    return getLineNumberOfStatement(match);
}
/**
 * Description of the Method
 *
 * @param linum Description of the Parameter
 */
public void advanceToLineNumber(int linum)
{
    // If the line number is out of range, then obviously we cannot advance
    // the file pointer
    if (linum <= 0 && linum > totalNumOfLines)
    {
        return;
    }

    try
    {
        currentTextIndex = 1;

        // Go ahead and read the specified number of lines to get back
        // in synch
        while (currentLineNumber < linum)
        {
            currentText = bufferedReader.readLine();
            currentLineNumber++;
        }
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

/**
 * Gets the statement line number based on the argument. What we are
 * trying to do here is find the statement within the source code that
 * matches what we are looking for. If we have a match, then return the
 * line number of the source code statement that matched what we needed.
 *
 * @param textToFind This is the text we need to find within the
 * source code
 * @return The line number of the statement within the source
 * code where the match was found.
 */
public int getLineNumberOfStatement(String textToFind)
{
    while (currentText != null)
    {
        // First check within the line to see if we can find a match
        currentTextIndex = currentText.indexOf(textToFind,
            currentTextIndex);
        if (currentTextIndex != -1)
        {
            // A match has been found, return the line number
            // But first, move the text pointer so we won't process the
            // same string
            currentTextIndex++;
        }
    }
}

```



```

        return currentLineNumber;
    }

    try
    {
        // If a match is not found on this line, get an entirely new
        // line. Should actually use JavaCC here ...
        currentText = bufferedReader.readLine();
        if (currentText == null)
        {
            break;
        }
        // Set the pointer to the beginning of the line
        currentTextIndex = 0;
        // Increment to the next line
        currentLineNumber++;
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

// If we get to here, then we have failed somehow ...
return -1;
}

/**
 * Closes the File I/O streams.
 */
public void close()
{
    try
    {
        bufferedReader.close();
        fileReader.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
}

```

E-20 STREAMREDIRECTTHREAD.JAVA

```
/*
 * @(#)StreamRedirectThread.java 1.3 01/12/03
 *
 * Copyright 2002 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */
/*
 * Copyright (c) 1997-2001 by Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
 * modify and redistribute this software in source and binary code form,
 * provided that i) this copyright notice and license appear on all copies of
 * the software; and ii) Licensee does not utilize the software in a manner
 * which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind. ALL
 * EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING
 * ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT
 * BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING,
 * MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL
 * SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR
 * DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 * HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF
 * THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF
 * THE POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control of
 * aircraft, air traffic, aircraft navigation or aircraft communications; or
 * in the design, construction, operation or maintenance of any nuclear
 * facility. Licensee represents and warrants that it will not use or
 * redistribute the Software for such purposes.
 */

package simple.util;

import java.io.*;

/**
 * StreamRedirectThread is a thread which copies it's input to it's output and
 * terminates when it completes.
 *
 * @author Robert Field
 * @created April 7, 2002
 * @version
 * @(#) StreamRedirectThread.java 1.3 01/12/03 00:15:38
 */
public class StreamRedirectThread extends Thread
{
    private final Reader in;
    private final PrintStream out;
    private final static int BUFFER_SIZE = 2048;
    private char[] cbuf = new char[BUFFER_SIZE];

    /**
     * Constructor for the StreamRedirectThread object
     *
     * @param name Description of the Parameter
     * @param in Description of the Parameter
     * @param out Description of the Parameter
     */
}
```

```

public StreamRedirectThread(String name, InputStream in, PrintStream out)
{
    super(name);
    this.in = new InputStreamReader(in);
    this.out = out;
    setPriority(Thread.MAX_PRIORITY - 1);
}

/**
 * Copy.
 */
public void run()
{
    try
    {
        int count;
        while ((count = in.read(cbuf, 0, BUFFER_SIZE)) >= 0)
        {
            String output = new String(cbuf, 0, count);
            out.print(output);
            output = null;
        }
        out.flush();
    }
    catch (IOException exc)
    {
        System.err.println("Child I/O Transfer - " + exc);
    }
}
}

```

E-21 UPDATETIME.JAVA

```
package simple.fault;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

import simple.util.*;
import java.util.*;

/**
 * This class represents a breakpoint action where time is updated
 */
/**
 * @author Neil Acantilado
 * @author Chris Acantilado
 * @created April 27, 2002
 */
public class UpdateTime extends PrimitiveFault
{
    /**
     * Constructor for the FieldFault object
     */
    public UpdateTime()
    {
        super(SimpleTrek.SIMPLE_CLIENT_CLASS, SimpleTrek.SIMPLE_SOURCE_LINE,
            "currentTime");
    }

    /**
     * Will update time on the SimpleHelper on the target JVM
     */
    /**
     * @param vm The target virtual machine
     * @param frame The frame the breakpoint was invoked in
     * @param objectReference The objectReference. Is null if static.
     * @param currentTime The currentTime
     * @param thread The thread the breakpoint was invoked in
     */
    public void execute(ThreadReference thread, VirtualMachine vm,
        StackFrame frame, ObjectReference objectReference, long currentTime)
    {
        // Process class local elements ...
        try
        {
            // Search for the local ...
            LocalVariable local = frame.visibleVariableByName(variableName);
            if (local == null)
            {
                return;
            }

            // Set the value and update numOfIterations counter
            frame.setValue(local, vm.mirrorOf(currentTime));
        }
        catch (Exception e)
        {
            // Something happened ... ignore for now
        }
    }
}
```

E-22 UTIL.JAVA

```
package simple.util;

import com.sun.jdi.*;
import com.sun.jdi.request.*;
import com.sun.jdi.event.*;
import com.sun.tools.example.debug.expr.*;
import com.sun.tools.example.debug.bdi.*;

/**
 * Description of the Class
 *
 * @author      nacantil
 * @created     April 27, 2002
 */
public final class Util
{
    private final static String boolType = "boolean";
    private final static String intType = "int";
    private final static String doubleType = "double";
    private final static String floatType = "float";
    private final static String longType = "long";
    private final static String stringType = "java.lang.String";

    /**
     * Description of the Method
     *
     * @param type Description of the Parameter
     * @param vm Description of the Parameter
     * @return Description of the Return Value
     */
    public static Value createRandomValue(String typeName, VirtualMachine vm)
    {
        if (typeName == null)
        {
            return null;
        }

        if (typeName.equals(boolType))
        {
            boolean randBool = false;
            if (Math.random() > 0.5)
            {
                randBool = true;
            }
            return vm.mirrorOf(randBool);
        }

        if (typeName.equals(intType))
        {
            int randInt = (int) (Math.random() * 1000.0);
            if (Math.random() > 0.5)
            {
                randInt *= -1;
            }
            return vm.mirrorOf(randInt);
        }

        if (typeName.equals(doubleType))
        {
            double randDbl = (double) (Math.random() * 1000.0);
            if (Math.random() > 0.5)
```

```

        {
            randDbl *= -1.0;
        }
        return vm.mirrorOf(randDbl);
    }

    if (typeName.equals(floatType))
    {
        float randFlt = (float) (Math.random() * 1000.0);
        if (Math.random() > 0.5)
        {
            randFlt *= -1.0f;
        }
        return vm.mirrorOf(randFlt);
    }

    if (typeName.indexOf(stringType) != -1)
    {
        double garbage = (float) (Math.random() * 1000.0);
        return vm.mirrorOf(String.valueOf(garbage));
    }

    System.out.println("Error in createRandomValue: " + typeName +
        " not currently supported.");

    return null;
}

/**
 * Description of the Method
 *
 * @param type      Description of the Parameter
 * @param valToSet  Description of the Parameter
 * @param vm        Description of the Parameter
 * @return          Description of the Return Value
 */
public static Value createValue(String typeName, String valToSet,
    VirtualMachine vm)
{
    if (typeName == null)
    {
        return null;
    }

    if (typeName.equals(boolType))
    {
        return vm.mirrorOf(Boolean.valueOf(valToSet).booleanValue());
    }

    if (typeName.equals(intType))
    {
        return vm.mirrorOf(Integer.parseInt(valToSet));
    }

    if (typeName.equals(doubleType))
    {
        return vm.mirrorOf(Double.parseDouble(valToSet));
    }

    if (typeName.equals(floatType))
    {
        return vm.mirrorOf(Float.parseFloat(valToSet));
    }
}

```

```

        if (typeName.equals(longType))
        {
            return vm.mirrorOf(Long.parseLong(valToSet));
        }

        if (typeName.indexOf(stringType) != -1)
        {
            return vm.mirrorOf(valToSet);
        }

        System.out.println("Error in createValue: " + typeName +
            " not currently supported.");

        return null;
    }
}

```

E-23 UTILITYASPECT.JAVA

```
package simple.aspect;

import java.util.*;
import simple.*;
import simple.fault.*;

/**
 * This is an aspect that updates the time for the event-thread
 *
 * @author      Neil Acantilado
 * @author      Chris Acantilado
 * @created     April 7, 2002
 * @version
 */
privileged aspect UtilityAspect
{
    private EventThread eventThread = null;
    private Calendar calendar = Calendar.getInstance();

    // Advice will get a refernce to the EventThread instance
    after(EventThread eventThread):
        target(eventThread) && execution(EventThread.new(..))
    {
        this.eventThread = eventThread;
    }

    // Advice will reset the eventThread startTime whenever the execute method
    // of a StartTime instance is invoked.
    after(): execution(* StartTime.execute(..))
    {
        eventThread.setStartTime();
    }
}
```


THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, VA
2. Dudley Knox Library
Naval Postgraduate School
Monterey, CA
3. Professor Bret Michael
Naval Postgraduate School
Monterey, CA
4. Professor Richard Riehle
Naval Postgraduate School
Monterey, CA